Titre: Title:	Implementing mobility for large scale analysis and optimization application
Auteur: Author:	Qun Zhou
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Zhou, Q. (2003). Implementing mobility for large scale analysis and optimization application [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7229/

UNIVERSITÉ DE MONTRÉAL

IMPLEMENTING MOBILITY FOR LARGE SCALE ANALYSIS AND OPTIMIZATION APPLICATION

QUN ZHOU
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
AUGUST 2003

Your file Votre référence
ISBN: 0-612-89221-2
Our file Notre référence
ISBN: 0-612-89221-2

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

IMPLEMENTING MOBILITY FOR LARGE SCALE ANALYSIS AND OPTIMIZATION APPLICATION

présenté par: ZHOU QUN

en vue de l'obtention du diplôme de: Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

M. OZELL Benoît, Ph.D., président

M. GUIBAULT François, Ph.D., membre et directeur de recherche

M. PIERRE Samuel, Ph.D., membre

To my parents, my wife and my new born daughter

ACKNOWLEDGEMENTS

Thanks very much to my director, Francois Guibault, who gave me very important advices when I met difficulties in the research, guided me to the right direction when I made mistakes in the project, and helped me to write the French part of this document. I appreciate very much his efforts and contributions on my studying.

Thanks very much to professor Benoit Ozell and professor Samuel Pierre, who spent their precious time to read my thesis and gave out valuable suggestions.

Thanks very much to Mr. Jean-Yves Trépanier who gave me both spiritual and financial support in the project.

Thanks very much to my mother-in-law Yuan Li, and my wife Feng Shen. They tried their best to take all the chores when I was very busy in studying and researching.

Thanks to my colleagues, Bin Chen, DaoJun Liu, Penney Wang and YuanLi Wang, who worked with me and brought me the happy days in CERCA.

RÉSUMÉ

Les modèles Client/Serveur et les Agents Mobiles constituent deux approches pour l'implantation de code mobile pour des applications distribuées déployées à grande échelle. En comparant le modèle des Agents mobiles, une technologie emmergente, à celui du Client/Serveur, on constate que ce dernier est un modèle plus mature, et plus largement utilisé par les développeurs. Il est en outre primordial d'adopter un standard de communication uniforme qui soit compatible avec toutes les plate-formes de calcul utilisées. Les deux approches citées présentent chacune des avantages et des inconvénients, qui ont été analysés dans le cadre du projet VADOR et de ce mémoire.

Une partie du système VADOR, le "CPU Server", a été implanté en utilisant chacune des approches. Avec le modèle Client/Serveur, les serveurs du systèmes Vador communiquent entre eux en passant des séries de codes prédéfinis selon un protocole propriétaire. La caractéristique la plus importante du "CPU Server" est d'effectuer le transfert d'information d'une façon sure et efficace. D'autre part, le "CPU server" maintient une structure de données complexe pour le traitement et la synchronisation des tâches déclenchées par l'"Executive server" sur des fils d'exécution distants.

Dans sa version basée sur les Agents Mobiles, le "CPU Server" existe sous l'apparence d'un groupe d'agents. L'"Executive Server" distribue les tâches en envoyant le code exécutables plutôt que le code statique à n'importe quelle machine disponible sur le réseau qui sera capable d'exécuter les applcations. Par l'utilisation d'un exemple réel issu d'un calcul d'ingénierie, ce mémoire présente le processus détaillé pour décrire et évaluer la performance de chacune des deux approches pour l'accomplissement d'une même tâche.

La solution Client/Serveur travaille d'une façon robuste et stable, mais elle n'est pas facile à déployer et à entretenir, elle ne fournit par ailleurs pas de méthodes qui permettent d'éviter les effets de congestion du réseau. La solution basée sur les Agents Mobiles offre suffisamment de mobilité, d'autonomie et d'intelligence pour réaliser efficacement les tâches demandées, cependant la technologie de sa platforme nécessite plus de développement et de standardisation.

ABSTRACT

The *Client/Server* and *Mobile Agents* models are two approaches to implement code mobility for large scale applications in a distributed system. Compared to the *Mobile Agents* model that is emerging technology, the *Client/Server* is very mature, and it is used widely by the software developers and has gotten plenty of experience. It is still necessary to set up a global communication standard for all those platforms. Both of the above two approaches have the advantages and disadvantages that have been analyzed in the *VADOR* project. The *CPU server* is implemented by both of the two approaches. Under the *Client/Server* solution, servers communicate each other by passing a serial of static codes whose structure is predefined according to a certain protocol. A most important thing of the *CPU server* is to make the information transfer accurate, brief and scalable. On the other hand, the *CPU server* maintains a complicated agency to process all kinds of jobs which come from the *Executive server* or remote threads. Under the *Mobile Agents* solution, the *CPU server* exists in appearance of group of agents. The *Executive server* commits tasks by sending the executable codes rather than the static codes to any available machines which is powerful enough to carry the applications. Using a real example of engineering calculation, this thesis presents the detailed process to describe how the two appraches perform in accomplishing the same task. The *Client/Server* solution works robustly and stably, but it is not easy for maintaince and deployment, it is also lack of methods to avoid the "bottleneck" effect of the network. The *Mobile Agents* solution has sufficient mobility, autonomy and intelligence to finish the tasks efficiently, however, the technology of its platform needs further development and standardization.

CONDENSÉ EN FRANÇAIS

Introduction

La conception basée sur des méthodes numériques d'analyse est un domaine qui a évolué très rapidement au cours des vingt dernières années. Des technologies matures pour la simulation de systèmes physiques complexes sont maintenant disponibles dans la grande majorité des disciplines du génie. Dans ce contexte, le projet VADOR vise à développer une infrastructure logicielle permettant d'intégrer des applications d'analyse en ingénierie provenant de différentes disciplines. Le système VADOR cherche ainsi à fournir aux ingénieurs concepteurs un environnement multidisciplinaire d'exécution des applications d'analyse intégrant des modèles à différents niveaux de fidélité et capable de gérer les résultats des analyses produits au cours de la phase de conception préliminaire de grands systèmes, particulièrement en aéronautique.

VADOR vise à intégrer non seulement des applications commerciales, mais également un grand nombre d'applications patrimoniales, afin de construire des processus complexes d'analyse utilisant différentes applications dans un environnement de calcul distribué et hétérogène. L'objectif principal du présent mémoire est d'étudier différentes approches de distribution des tâches liées à l'exécution des applications d'analyse, et particulièrement de comparer les approches clients/serveurs (C/S) et les approches à base d'agents mobiles (AM). Une fois l'approche la plus prometteuse identifiée en fonction des besoins du projet VADOR, une implantation complète de l'approche retenue sera intégrée à l'environnement de simulation.

L'architecture proposée pour le projet VADOR découpe les fonctionnalités de ges-

tion des données et d'exécution des tâches en un nombre restreint de serveurs dédiés. Chaque serveur est responsable d'un aspect fondamental de la gestion de l'environnement d'analyse. Les tâches liées à l'exécution des processus de simulation ont été confiées à un serveur central nommé *Executive Server*, qui délègue l'exécution de chacune des tâches à des serveurs de calcul nommés *CPU Servers*. Deux approches permettant cette délégation de l'exécution ont été étudiées, soit une délégation uniquement basée sur une approche C/S, et une délégation utilisant une approche par AM.

Dans l'approche C/S, l'*Executive Server* agit comme un client qui envoie des requêtes au *CPU Server*, afin d'utiliser des services de gestion de fichiers et d'exécution d'applications. La méthode de communication entre les deux ordinateurs est alors basée sur les *sockets*, qui permettent d'acheminer les requêtes d'un ordinateur à l'autre, en les encapsulant sous forme de paquets.

Dans l'approche basée sur les AM, le *CPU Server* est constitué d'un ensemble d'agents mobiles qui s'exécutent de façon autonome. Lorsque l'*Executive Server* doit utiliser une ressource d'un ordinateur distant, une série d'agents sont envoyés au travers du réseau en utilisant les capacités de communication de la plate-forme agent.

Étant donné que chacune des approches présente des avantages et des inconvénients, les concepteurs de l'environnement VADOR doivent être en mesure d'exercer un choix éclairé pour la réalisation de la version du système qui sera déployée et mise en production. L'objectif de ce mémoire est donc de comparer les deux approches dans le contexte d'une application concrète impliquant un nombre représentatif de tâches et de machines.

Les Approches de Distribution de l'Exécution

L'approche Client/Serveur

L'approche C/S est une forme de distribution des calculs selon laquelle un programme s'exécutant sur une machine communique avec un programme s'exécutant sur une autre machine afin d'échanger de l'information. Le client agit comme un émetteur de requêtes pour l'utilisation de services, alors que le serveur est défini comme le fournisseur des services. Cette distinction entre le client et le serveur est strictement logicielle, il n'existe en effet souvent aucune différence matérielle entre la machine client et la machine serveur.

Les architectures client/serveur (AC/S) sont présentées comme une façon de contourner les limites imposées par des approches centralisées du type *File Sharing Architectures*, qui ont été démontrées n'être utilisables que dans les cas de faible utilisation des ressources, de faible compétition, et de volumes de données restreints. Le concept de niveau (*tier*) fournit une méthode pratique permettant de classifier les différentes AC/S. Les deux types d'architectures de type AC/S les plus courantes sont les architectures à deux et trois niveaux.

Il est maintenant admis que l'architecture à trois niveaux, qui distingue les aspects d'interface usager, de traitements liés au domaine d'application et de traitements des données, présente d'excellentes possibilités de mise à l'échelle, et peut supporter, pour certains types d'applications, des centaines d'usagers. Il n'en demeure pas moins que plusieurs aspects délicats liés à la conception d'un système C/S doivent toujours être étudiés attentivement par les développeurs. Un premier aspect qui exige une conception soignée consiste à gérer correctement les aspects de concurrence au niveau des serveurs, lorsqu'un grand nombre de clients veulent

partager des ressources distribuées. Un second aspect est la sécurité, qui comprend en fait deux défis: le transfert de données sensibles de façon sécuritaire entre le client et le serveur, et l'authentification des clients. Un dernier aspect délicat de l'approche C/S est lié à la croissance du nombre de clients et de ressources qui doit être gérée de façon à ce que l'ensemble du système reste performant.

L'approche par Agents Mobiles

Un agent mobile est défini comme une composante logicielle indépendante qui agit de façon autonome au nom d'un usager. Un agent mobile incorpore deux caractéristiques essentielles. D'abord l'autonomie, qui permet à l'agent d'agir par lui-même en ayant recours aux données et à la logique qui lui sont associées, et sans nécessité d'intervention humaine. La seconde caractéristique est la mobilité de l'agent. La mobilité permet à l'agent de voyager vers un hôte sur lequel sont physiquement stockées les données nécessaires à son exécution. Il y a également certaines considérations techniques liées au développement de solutions à base d'agents qui devraient être considérées: les caractéristiques de mobilité, de sécurité et de standardisation des méthodes de communication sont parmi les plus importantes.

On distingue trois types de mobilité pour un agent. La première, la mobilité contrainte, s'apparente aux approches C/S, dans la mesure où le code est déplacé vers la machine cible, mais que le déclenchement de l'exécution du code est initiée à distance par un *Client* qui accède ainsi au ressource de l'hôte. Le second type de mobilité est la mobilité faible, où migration non transparente, qui propose un mode de déplacement du code dans lequel une partie des données peut être transférée lors de la migration de l'agent, mais où l'état d'exécution de l'agent, au moment de déclencher la migration, n'est pas conservé. Le code est autonome et sera utilisé pour réinitialiser le processus d'exécution une fois que l'agent aura atteint

sa destination. Le troisième type de mobilité, appelé migration transparente ou mobilité forte, permet à une unité d'exécution de se déplacer comme un tout et de conserver son état d'exécution au cours du déplacement.

Les avantages de la mobilité contrainte s'apparentent à ceux des approches C/S au point de vue de la flexibilité et de la légèreté, mais souffrent d'un manque d'autonomie. La mobilité faible, quant à elle, permet d'implanter une réelle autonomie de l'agent. Enfin, la mobilité forte présente l'ensemble complet des caractéristiques associées aux approches à base d'agents, mais les mécanismes nécessaires pour supporter une telle approche entraînent des coûts importants en terme de développement et de maintenance.

La sécurité est un aspect critique des approches par AM. Les systèmes à base d'agents permettent à des segments de code de se déplacer relativement librement et de s'exécuter sur différents noeuds d'un réseau. Cette mobilité peut parfois entraîner des problèmes importants de sécurité. Trois types d'attaques sont à craindre: les attaques d'un agent malicieux envers sa plate-forme d'exécution, les attaques d'un agent malicieux envers d'autres agents s'exécutant sur la même plate-forme et les attaques d'une plate-forme malicieuse envers les agents qu'elle exécute. Afin de contrer ces différents types d'attaques, des recherches sont actuellement menées qui tendent à indiquer qu'au moins quatre caractéristiques sont requises d'une plate-forme agent: l'authentification, la confidentialité, l'intégrité et l'encerclement (*sandboxing*).

Un autre aspect qui doit être considéré minutieusement lors de l'implantation d'une solution à base d'agent est le mécanisme de communication entre les agents. Cette communication inter-agent doit particulièrement être considérée dans le contexte où des agents sont exécutés par des processus logiciels séparés qui comptent sur une communication entre les agents pour arbitrer l'accès aux ressources et la coor-

dination des tâches à effectuer. La communication inter-agents pose le problème de la standardisation des langages de communication. Les approches les plus évoluées et répandues sont basés sur un ou plusieurs des standards suivants: Foundation for Intelligent Physical Agents (*FIPA*), the Knowledge Query and Manipulation Language (*KQML*), et le Mobile Agents System Interoperability Facility (*MASIF, OMG Group 1998*).

Exécution Distribuée d'applications Patrimoniales

Du point de vue fonctionnel, les deux approches de distribution présentées peuvent être utilisées pour réaliser les composantes logicielles associées au serveur de calcul (*CPU Server*). L'évaluation de ces deux approches doit donc se faire sur la base de la performance de chacune des approches, et sur leur simplicité de réalisation et d'utilisation.

L'objectif essentiel du *CPU Server* est de permettre l'exécution à distance d'applications patrimoniales. Ceci implique le déclenchement d'applications externes à partir d'un programme, en fournissant à chaque application certains paramètres sur la ligne de commande, et en particulier les chemins d'accès aux différents fichiers d'entrée et de sortie, sous forme de chaînes de caractères. L'une des tâches les plus complexes associée à cette fonctionnalité est la gestion des différents fichiers de données utilisés par chacune des applications, qui doivent être rendus accessibles de façon transparente aux applications, quel que soit l'endroit sur le réseau où les fichiers sont initialement ou ultimement stockés.

Pour des raisons de sécurité et de standardisation, les concepteurs du projet VADOR ont choisi de réaliser les fonctions de gestion et de transfert de fichiers en utilisant le serveur web *Apache* auquel est associé le serveur *Tomcat* qui permettent collec-

tivement le transfert bidirectionnel de fichiers. Le *CPU Server* dispose ainsi des services nécessaires au téléchargement de fichiers, mais la responsabilité de coordination des transferts lui incombe.

En plus des tâches associées à l'exécution des applications patrimoniales, le *CPU Server* doit également accomplir certaines opérations liées au contrôle des processus d'exécution, au contrôle des fils d'exécution et au transfert des fichiers. Ces commandes internes sont classées en deux catégories: synchrones et asynchrones.

L'implantation Client/Serveur du *CPU Server*

Dans cette solution, les communications entre l'*Executive Server* et le *CPU Server* sont assurées à l'aide de *sockets*. La principale considération demeure le transport de l'information d'exécution qui doit être transférée de façon efficace et précise. Comme les tâches d'exécution vont des tâches très simples aux tâches très complexes, un protocole général de transmission de l'information d'exécution doit être développé, qui permette de transférer des informations d'exécution comprenant un nombre arbitraire de sous-tâches. Le temps requis pour exécuter chacune des sous-tâches qui composent une tâche d'exécution peut varier considérablement (de la seconde à la semaine), et toutes les sous-tâches sont amalgamées dans un seul ordre d'exécution au niveau de l'*Executive Server*. Cet ordre d'exécution est transféré sous la forme d'un objet unique vers le *CPU Server* au cours d'une seule transaction, et la connexion entre les serveurs est ensuite coupée.

Du côté de l'*Executive Server*, les sous-tâches sont classifiées comme `EXTERNAL` ou `INTERNAL`, et chaque fichier de donnée est classifié comme `INPUT-FILE`, `OUTPUT-FILE` ou `OPTIONAL-FILE`. Ces informations sont ensuite encapsulées dans une structure de données d'échange à l'aide de la classe *WrapperCommandBuilder*, puis la struc-

ture de données est transférée au *CPU Server* à l'aide de la class *WrapperProxy*.

Du côté du *CPU Server*, la classe *WrapperCommandInterpretor* reçoit et interprète les commandes d'exécution, puis transfère cette information à la classe *WrapperJobProducer* pour qu'elle soit traduite en une série de tâches effectuant le travail réel, sous le contrôle du *CPU Server*.

Un autre défi dans l'exécution des tâches réside dans le contrôle du flot d'exécution. Afin d'économiser le temps d'exécution sur le processeur, les tâches qui peuvent s'exécuter de façon indépendante devraient être exécutées simultanément dans des fils d'exécution séparés. Par ailleurs, l'obtention des résultats d'exécution de certaines sous-tâches constitue des pré-conditions à l'exécution de certaines autres sous-tâches, qui doivent donc attendre que l'exécution des premières tâches soit complétée avant d'être déclenchées. Les classes *WrapperSubCmdJob*, *WrapperJobGroup* et *WrapperSubJob* servent donc à organiser l'exécution des tâches soit de façon parallèle ou soit de façon séquentielle.

Comme plusieurs tâches peuvent s'exécuter simultanément sur un même *CPU Server*, et que chaque tâche peut comprendre des sections d'exécution séquentielles et d'autres parallèles, un mécanisme de fonctions de rappel a été implanté au niveau du *CPU Server*. Le serveur maintient ainsi trois structures de données: le *ThreadsControler* qui gère les mécanismes de communication entre tous les fils d'exécution actifs d'une tâche sur un serveur; le *TaskControler* qui fournit un mécanisme centralisé de manipulation des informations sur toutes les tâches en cours d'exécution; et le *ResultsControler* qui enregistre et rend disponibles les résultats de chacune des sous-tâches. C'est sur la base des résultats accumulés par le *ResultsControler* que le *CPU Server* est en mesure de décider si une tâche devrait passer à l'étape suivante ou être annulée.

L'implantation à base d'agents mobiles du *CPU Server*

Cette solution tente de mettre à profit le plus grand nombre possible d'avantages annoncés par la technologie des AM afin d'implanter un prototype fonctionnel du *CPU Server*. Cette version du *CPU Server* suppose que chacune des machines du réseau sur lequel sera déployé le système VADOR est équipé d'une plate-forme d'AM. L'acteur en charge de l'exécution d'une tâche n'est plus un ensemble de programmes stationnaires installés sur chaque machine, mais bien un groupe d'agents créé spécifiquement pour la tâche. Afin de garantir un environnement de travail adéquat aux agents, une plate-forme stable et standard est absolument nécessaire. Celle-ci doit créer, interpréter, exécuter, transférer et détruire les agents.

Il existe actuellement deux principaux standards pour la spécification de systèmes à base d'agent. Le premier standard est basé sur la spécification *MASIF*, qui vise à promouvoir l'interopérabilité et la diversité des systèmes AM. La spécification *MASIF* recommande fortement l'utilisation de services CORBA comme base d'implantation pour l'infrastructure de communication entre agents. Le second standard est le FIPA, qui vise spécifiquement l'interopérabilité des agents et des systèmes agents entre les plates-formes de différents fournisseurs.

Le choix de plate-forme s'est porté vers le système *Grasshopper* qui implante tant le standard *MASIF* que le standard *FIPA*, en plus d'être disponible gratuitement pour la recherche.

Dans la solution à base d'agents, une seule copie du *CPU Server* est déployée sur la même machine que l'*Executive Server*. Selon le contenu précis de la tâche qui doit être exécutée, le *CPU Server* prépare une série de prototypes pour les différents types d'agents nécessaires à l'exécution de la tâche. L'exécution de la tâche elle-même comprend trois phases distinctes: d'abord, la phase de préparation de la

tâche, qui utilise la classe *WrapperTaskDeployer* pour rassembler les sous-tâches en groupes selon leur ordre d'exécution. Deuxièmement, la phase de construction des agents durant laquelle le *CPU Server* recrute deux types d'agents, les "soldats" et les "commandants". Chaque "soldat" traite une seule sous-tâche, alors que le "commandant" contrôle la cédule d'exécution globale et le processus d'exécution. Troisièmement, la phase de migration et d'exécution des agents, durant laquelle le *CPU Server* envoie les agents qu'il a préparés sur les plates-formes distantes afin d'exécuter les tâches de façon asynchrone. Si certains fichiers de résultats doivent être transférés sur des machines différentes de l'endroit où le résultat a été produit, des "soldats" sont envoyés sur les machines de destination afin de procéder au téléchargement des fichiers.

Application et Résultats

Afin de comparer l'efficacité, la robustesse et la facilité d'utilisation des deux approches, un processus de simulation a été implanté en utilisant chacune des méthodes de distribution. L'application choisie est une séquence de programmes servant à optimiser des profils d'ailes d'avion, afin d'en améliorer les performances aérodynamiques. Une portion importante de ce processus d'optimisation vise à approximer chaque profil à l'aide de courbes NURBS tout en utilisant un nombre minimal de points de contrôle pour une tolérance donnée.

Plan de Test

En vue de quantifier la performance lors de l'exécution du processus d'optimisation en utilisant chacune des approches de distribution, deux aspects ont été considérés. Le premier est le temps d'exécution, qui est le temps total entre le moment où la

première commande est envoyée de l'*Executive Server* et le moment où le dernier signal *NotifyFinish* est reçu du *CPU Server*. Le second aspect considéré est le volume total de transactions sur le réseau engendré par chaque approche.

Afin d'obtenir des statistiques fiables, des tests pour 26 configurations différentes des données sur les profils d'ailes ont été effectués, et pour chaque configuration, quatre séries de tests ont été effectuées, pour différents cas de distribution. Les deux premières séries de tests ont été effectuées en utilisant l'approche C/S, dans un cas, en plaçant tous les fichiers de données sur la même machine que le *CPU Server*, et dans le second cas, en plaçant les fichiers de données sur une machine différente. Les deux autres séries de tests ont été effectuées en utilisant l'approche par AM, encore une fois, pour des fichiers de données locaux dans le premier cas, et sur une machine séparée dans le second.

Comparaison des approches

Au niveau du transfert des données de contrôle des tâches, l'approche C/S doit prévoir des mécanismes d'encapsulation et d'interprétation des données qui permettent un échange en un nombre très limité de transactions (idéalement une seule) entre le client et le serveur. Dans le cas de l'approche par AM, ces mécanismes ne sont pas nécessaires, puisqu'ils sont pris en charge directement par la plate-forme, ce qui simplifie le développement du code.

Au niveau de l'exécution des tâches, le modèle C/S du *CPU Server* utilise un système complet de contrôle des différentes étapes et des résultats d'exécution. Dans l'approche par AM, le système de contrôle est légèrement simplifié. Chaque tâche est assignée à un agent *soldat*, qui sont collectivement coordonnées par un *commandant*. Le code par AM est ainsi, encore une fois, plus concis que celui par

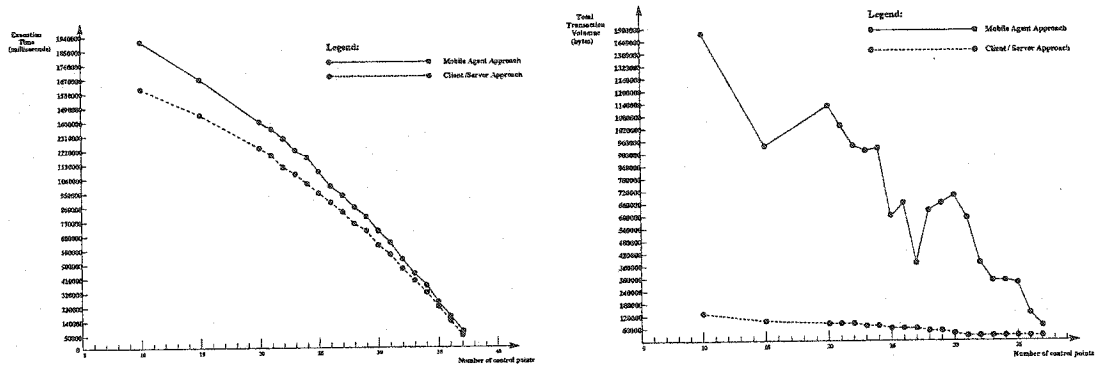
C/S.

Au niveau du déploiement et de la maintenance, le modèle C/S impose que chaque machine qui veut agir comme *CPU Server* reçoive sa propre copie du code serveur, et que l'exécution de ce code soit lancée comme un service local par le système d'exploitation de chaque machine. En terme de maintenance, cette approche implique que les nouvelles versions du code du *CPU Server* doivent être redéployées sur toutes les machines. Dans le cas de l'approche par AM, c'est la plate-forme agent qui est déployée sur chaque machine. Bien que ce ne soit pas le cas pour le moment, on peut envisager que dans un avenir plus ou moins proche, les systèmes d'exploitation puissent inclure une plate-forme agent. Le redéploiement d'une nouvelle version du *CPU Server* n'impliquerait alors qu'un minimum de travail, étant donné qu'une seule copie du code agent serait déployée sur la machine exécutant l'*Executive Server*.

Comparaison des résultats

Tant les tests simples que les tests plus complexes utilisés pour comparer les deux approches de distribution ont présenté des caractéristiques de performance similaires. La figure 1 présente les résultats obtenus lors de l'exécution d'une séquence de 26 processus complets d'optimisation utilisant chacune des deux approches de distribution du code. Les processus utilisés dans chacune des deux approches sont identiques et impliquent l'exécution de cinq applications servant collectivement à optimiser un profile d'aile d'avion.

Au niveau de la charge sur le réseau, pour le type d'application envisagé, le modèle C/S présente un avantage très net par rapport aux AM, dans la mesure où l'approche C/S ne transmet qu'une quantité limitée de données de configuration



(a) Temps d'exécution (b) Volume de trafic

Figure 1 Comparaison du Temps d'exécution et du volume de trafic pour les approches C/S et AM

contenue dans un vecteur, qui est ensuite interprétée par le *CPU Server*. L'approche par AM, de son côté, doit migrer non seulement les données de configuration, mais également une partie du code exécutable. Dans la situation analysée ici, il est estimé que l'approche par AM transfère entre 5 et 10 fois plus de données que son équivalent par C/S. Au niveau de l'efficacité, l'approche C/S se comporte également mieux que la méthode basée sur les AM. Pour chaque cas test analysé, les temps d'exécution associés à l'approche C/S sont systématiquement plus petits que ceux obtenus par l'approche AM.

Par ailleurs, au niveau de la stabilité, l'approche C/S est apparue beaucoup plus robuste que l'approche par AM. Plusieurs tests qui se sont exécutés sans difficulté en utilisant le modèle C/S n'ont pu être complétés à l'aide de la version par AM. D'autre part, les quantités de données communiquées entre plates-formes agent sont plus difficiles à prévoir que leur contrepartie par C/S comme en témoigne la figure 1b, où la courbe du volume de transfert de données est beaucoup plus monotone pour le modèle C/S que pour le modèle AM.

L'approche par AM conserve toutefois un certain nombre d'avantages sur la version C/S. En particulier, l'approche par AM permet une meilleure distribution des

responsabilités, en réduisant la quantité de travail effectuée par la machine exécutant l'*Executive Server*. D'autre part, l'approche par AM permet de simplifier l'architecture du code déployé au niveau du *CPU Server* en éliminant la nécessité d'implanter des structures de données et du code pour gérer l'exécution de chaque sous-tâche. Enfin, l'approche par AM devrait permettre de simplifier le déploiement et les mises à jour.

Il reste qu'en raison de sa maturité et de l'expérience acquise dans son utilisation par des générations de concepteurs, c'est l'approche C/S qui a été retenue comme base pour l'implantation de la version de production du *CPU Server* dans le système VADOR.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF NOTATIONS AND SYMBOLS

ACL	Agent Communication Language
CC	Communication channels
CERCA	CEntre de Calcul en Recherche Appliqué
CORBA	Common Object Request Broker Architecture
C/S	Client/Server
CSA	Client/Server Architecture
DBMS	DataBase Management System
EJB	Enterprise Java Bean
FIPA	Foundation For Intelligent Physical Agent
FSA	File Sharing Architecture
GUI	Graphic User Interface
HTTP	HyperText Transport Protocol
KQML	Knowledge Query and Manipulation Language
KSE	Knowledge Sharing Effort
MA	Mobile Agent
MAS	Multi-Agent System
MASIF	Mobile Ahent System Interoperability Facility
MP	Message Passing
OMG	Object Management Group
OSF/DEC	Open Software Foundation / Distributed Computing Environment

QMP	Queued Message Processing
PM	Process Migration
RDA	Remote Data Access
REV	Remote EValuation
RPC	Remote Procedure Call
SQL	Structured Query Language
WWW	World Wide Web
URL	Uniform Resource Locator

LIST OF ANNEXES

CHAPTER 1

INTRODUCTION

Engineers' requirement

To complete projects involving many departments and disciplines, engineers in large companies rely on extensive numerical computations and experimental testings which involve both cooperative and individual work. Aeronautical design is a typical example of such a framework. Aeronautical design involves the interaction of a large number of highly specialized engineering fields, often represented as separate departments within a larger design and manufacturing organization. While separate, these departments nevertheless constantly need to share and exchange large amounts of numerical data, particularly in the course of developping analysis procedures carried by each department to assess design constraints and evaluate prototype performance. The production of this data thus involves elaborate analysis processes as well as interventions by field experts that stear computations and validate analysis output. In order to efficiently manage and store this information, engineers need to rely on an information system which can provide an integration solution of legacy analysis applications. Most legacy applications are custom made, in-house applications. Currently, engineers must spend large amounts of time and money in trying to make various applications communicate with each other. Once communication issues have been resolved, a second challenge in aeronautical design and analysis is to create an efficient environment which allows to implement multi-department analysis processes. (Ndiaye *et al.*, 2000)

The VADOR project

VADOR (Virtual Aircraft Design and Optimization fRamework) is an ongoing project carried in collaboration with Bombardier Aerospace (Trépanier *et al.*, 2000). Its goal is to develop a framework for integrating multi-disciplinary and multi-fidelity engineering analysis programs and for managing analysis results. It provides engineers from all departments with a uniform and structured technical environment, that allows them both to define and run analysis procedures, and make resulting analysis data available to colleagues within and across departments. One fundamental aspect of the Vador framework is its capacity to handle various types of analysis procedures through the encapsulation of legacy applications within the framework. This allows the definition of elaborate processes involving the distributed execution of a large number of applications and data migration of results across a highly heterogenous network of workstations.

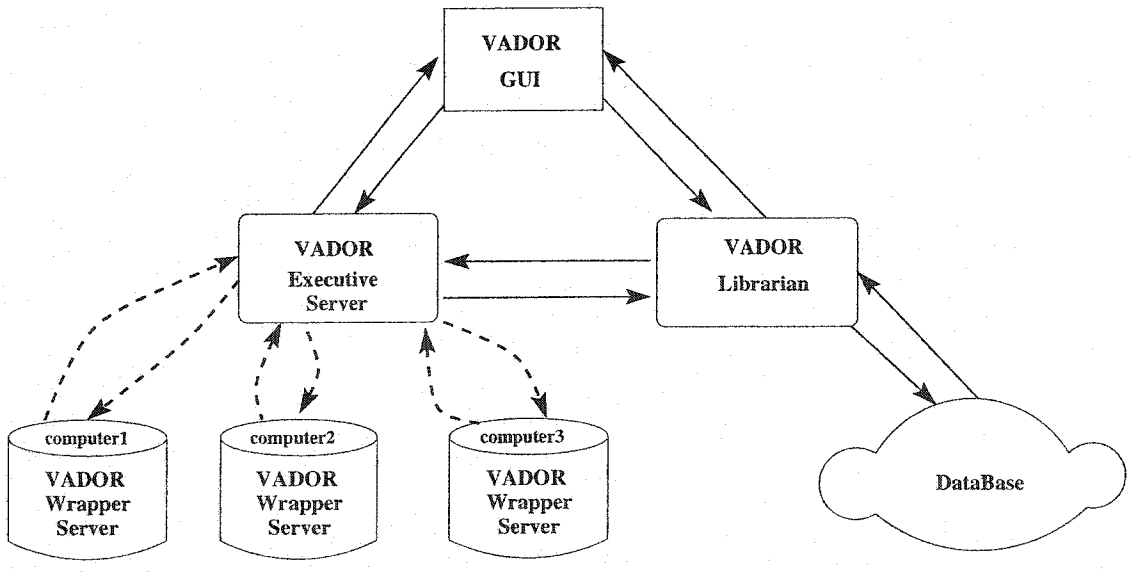


Figure 1.1 The global architecture of Vador

Figure 1.1 shows the overall VADOR architecture. Engineers interface with VADOR through the graphic user interface (GUI), client and all informations about legacy applications are encapsulated in strategy and data components that are stored and managed in the VADOR database system. The *Executive server* and the *CPU server*(Wrapper) play important roles in providing the framework with the ability to remotely launch analysis precesses. The *Executive server* acts as a management server that coordinates all remote task executions through a delegation approach to the *CPU server*, and the *CPU server* performs flexible and adaptable wrapping of in-house and commercial applications used by engineers. Another aspect of the functionality provided by VADOR is the *load-balancing* of tasks which is in charge of checking the current load status of host computers so that the whole system can optimize job distribution.

Legacy application distribution and task mobility

Mobility is a central concept of distributed computing. Strict policies guiding code execution among a group of distributed machines must be enforced in order to utilize resources of all types to finish the task as efficiently as possible. There are two main approaches that exist to code mobility: the distributed *Client/Server* and *Mobile Agents* approaches.

Client/Server model

The *Client/Server* (C/S) is an organization of computers on a network where one computer (the client) uses resources from another computer (the server) attached to the network through requests and responses. A *Client* is defined as a requester of services and a server is defined as a provider of services. The definition of *Client*

and *Server* is functional rather than physical, that is, a computer can act both as a *Server* and a *Client* depending on the software configuration.

The *C/S* approach is the main approach used in the VADOR project. As a *Client*, the *Executive server* sends requests to the *CPU server* to access file management and legacy application execution services. The *Executive server* also uses services from the *load-balancing* server to select the most suitable machine for the execution of a task. However, in view of an engineer submitting a task to the VADOR system, the *Executive server* acts as a *Server*. The communication method between two machines is based on sockets, and requests are encapsulated in message packages, that migrate from one computer to the other through sockets.

The *C/S* model has a relatively long history and is mature and stable enough for software development. There are many articles that elaborate on this approach. This model nevertheless has some limitations such as a possible bottleneck effect in transfering and treating messages. These aspects will be futher discussed in chapters two and three.

Mobile Agents model

The purpose of the *Mobile Agents*(MA) model is to migrate a piece of program (agent) to a remote computer for execution. An agent is a simple program which can help accomplish a task without a continued interaction with the user, it can exhibit some intelligent behavior involving learning, communicating and acting in complex environment.

In the VADOR framework, the *CPU server* and the *load-balancing* module can be managed by *MA* technology, since they always work autonomously on behalf of the engineers using the VADOR system. Depending on the capacity of a given

computer, there may be several tasks running on a single machine. Multi-Agent System(MAS) provides the mechanisms to coordinate all agents which encapsulate a related tasks. In a MAS, heterogeneous distibuted services are represented as agents which routinely use a particular communication language to interact with each other in order to coordinate and complete the tasks to which they have been assigned.

The most important property of an agent is mobility, because this property can make the deployment and execution procedures more flexible and efficient. Different from stationary agents, mobile agents are not bound to the system where they start their execution, rather they can transport themselves from one system to another over a network. This property makes distibuted systems based on mobile agent technology much easier to design, implement, and maintain. As mentioned, the *Executive server* can submit a single or a series of tasks (such as execution of legacy codes or collection of load-balancing information) to a specified machine by sending several *Wrapper* or *load-balancing* agents to it, and relying on the MAS running on that computer to realize the cooperation among agents in order to successfully accomplish a job . In a Muti-tasks mission that involves running many legacy applications, a *Wrapper* agent could migrate to a more suitable machine which hosts the target code and then migrate again to another one until all tasks are completed. Advantages of migration also appear when selecting a lightly-loaded computer while launching a task. Every time the *Wrapper* agent begins to run, it should contact all *load-balancing* agents running on remote machines to gather detailed loading status of the critical resource needed for the task at hand and migrate to the best computer after comparison. In such a scenario, the *Wrapper* optimizes it's own execution through "intelligent" behavior.

Objective

Actually, both *C/S* and *MA* approaches have advantages and disadvantages. Using the *C/S* model, software developers must consider a priori how the client and server parts of that system will be deployed on the computers, and how request/response packets will be serialized and interpreted. As for *MA*, the typical challenge is how to standardize the communication methods between agents and agent platforms produced by different software engineers. Besides, the performance is another aspect which affects the final choice of the software developer. In the *MA* approach, agent platform is the key component which guarantees the correct execution of each agent. If its technology is not sufficiently mature and efficient, the performance of the whole system will suffer. The *C/S* approach doesn't have this burden, it may display better performance in a production environment. Developers should balance the convenience and performance of both approaches and then make a decision.

Under different conditions, the developers must consider various strategies for their projects. They should make a choice between the *C/S* model and *MA* model based on many factors such as the efficiency, security, scalability, failure handling and convenience. The objective of this thesis is to study the merits and disadvantages of each model and give out a clear comparision in the context of a concrete application involving a large number of tasks and machines.

Organization of this document

In chapter 2, we will describe the background and current status of the *C/S and MA*, based on approaches including FIPA (OMG Group, 2001) standard and

mobile-agent technology. Chapter 3 will present a comparison between the approaches to mobility in the context of legacy applications. It provides a discussion of several issues such as communication protocol, implementation of mobility and security protection. Chapter 4 addresses our efforts to actually compare the two approaches to mobility in a real life system, VADOR. The final part is the conclusion and the future work.

CHAPTER 2

APPROACHES TO DISTRIBUTED COMPUTING

This chapter presents a comparison of the two dominant approaches to distributed computing, namely the *C/S* and *Mobile Agents* approaches, and discusses their respective merits in the context of the execution of legacy application.

2.1 Client/Server Networking

The concept of *C/S* first appealed in the 1980s when personal computers began to get connected to each other over networks. Much different from the "Mainframe architecture", the *C/S* architecture is a versatile, messaged-based and modular infrastructure that improves usability, flexibility, interoperability and scalability. Currently, the *C/S* model has become the most general and applicable solution to all types of computer transactions.

2.1.1 Overview of the Client/Server Model

The *C/S* model is a form of distributed computing where one program communicates with another (a *Client* contacts with a *Server*) in order to exchange information.

In this model, the *Client* acts as a requester of services whereas the *Server* is defined as a provider of services. Both of them handle particular responsibilities. The *Client* generally presents an interface to the users, translate the user's demands

into a desired protocol and then sends the request accross to the *Server*. After that, it keeps waiting until it gets the reponse from the *Server*, translates the message into "human-readable" results and then presents them to the user. The *Server*'s main job is to treat the requests from the *Clients*, interact with it's native database system, perform some computation and return the final result.

In a network, the *Client* and *Server* computers usually are two separate devices, each customized for their designed purpose. It is possible to create an interface that is independent of the *Server* which hosts the data. This also allows information to be stored in a central Sever machine and disseminated to different types of remote computers. Furthermore, since the User interface work is performed at the *Clients*' ends, the *Server* gets more computing resources to spend on analyzing queries and sending information. Therefore, this model provides a convenient way to interconnect programs that are distributed efficiently across different locations.

C/S networking focuses primarily on the applications rather than hardware, there is no apparent distinction between *Client* and *Server* devices. A computer can run both *Server* and *Client* programs, and alternatelly play the role of a *Server* to respond to other *Clients*, and moments later, behave as a *Client* to respond to another *Server*.

2.1.2 Introduction to Client/Server Architecture

The original PC network was based on the *File Sharing Architecture (FSA)*, where the *Server* downloads the files which are requested by users from a shared location to the user's desktop to run. This approach can be used under the conditions of low shared usage, low competition and low data volume(Trigon Blue Inc., 2000). The *Client/Server architecture (CSA)*(Software Engineering Institute (SEI), 1997a) is

presented to solve the limitations of FSA. In its typical mode of operation, a request made by a *Client* gets processed at the *Server* level rather than at the *Client* level, and it introduces a database *Server* instead of a file *Server*. Using for example a relational database system, users'queries can be replied to directly. CSA reduces network traffic by providing a query response rather than a complete file transport and thus increases performance. It also provides the fundamental framework that allows many technologies to plug in.

There are still many debates on the topic of how to construct a good *Client/Server architecture*. The most controversial issue revolves around the software tier issue. The *tier* concept provides a convenient way to group different classes of architectures. The two most popular tier architectures for CSA are *two-tiers* and *three-tiers*. But other special cases exist with *one-tier* models and others that are *N-tiers*.

2.1.3 Two-tier architecture (developped in the 1980s)

In this architecture, the two layers are: the *Client* (requester of services) and the *Server* (provider of services).

1. Structure and functions:

As illustrated in Figure 2.1, in a prototypical two-tier application, there is a very clear division between front end and back end tiers. The first tier, the *Client*, doesn't need to take care of data storage issues or processing multiple requests; the second tier, the *Server*, does no need to worry about user feedback and tricky user interface issues. In fact, there is another part: the protocol. The protocol bridges the gap between the *Client* and *Server* layers.

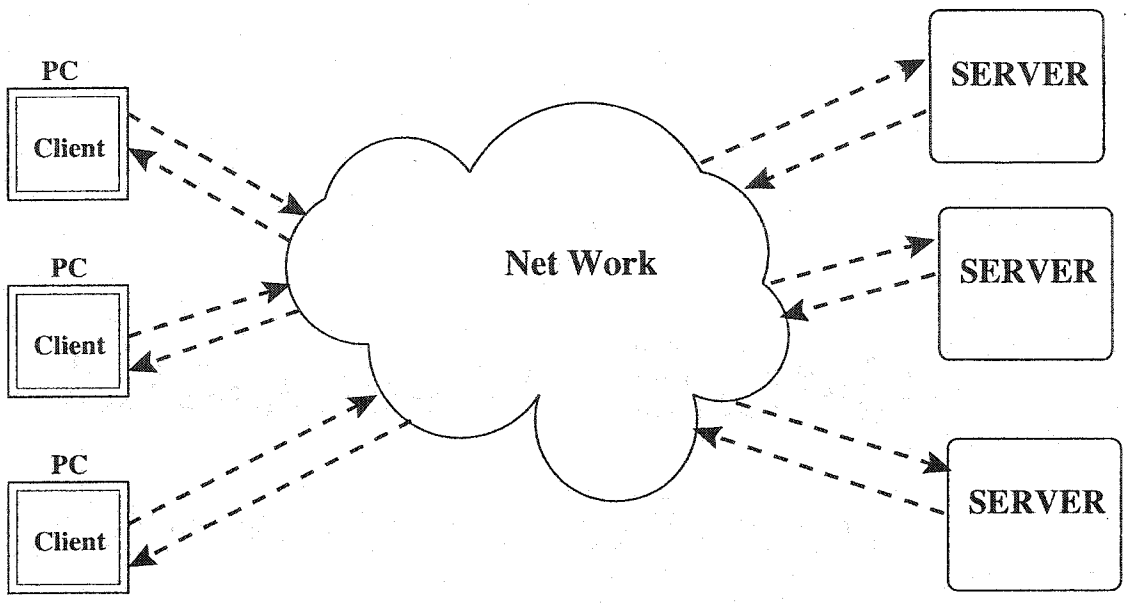


Figure 2.1 Two-tier architecture

The main functions of this architecture can be summarized in three components. The first one is the user interface which is traditionally located in the Client tier and usually written in a high-level language like Java or C++, it provides the services such as session management, text input and display management. The second function processes the management, it comprises process development, enactment, monitoring and resource services. Depending on the design approach, this function can be split and be incorporated either into the *Client* or the *Server* layer. The last function, database management, is always on *Server* side, and is in charge of data and file services.

2. Usage Considerations:

Since two-tier architectures have been built and used very early (since the mid 1980s), this design is well known and used throughout the industry, its technology is sufficiently mature.

If we want to develop a non-time critical information process where management and operations of the system are not complex, two-tier architecture is a good choice. This design is frequently used for decision support systems with light transaction loads. Two-tier architecture works very well in homogeneous environments with processing rules that do not change often, but it requires minimal operator intervention, the best workgroup size is expected to be less than 100 users (Software Engineering Institute (SEI), 1997c).

3. Merits and shortcomings:

In the two-tier *C/S* program, there is a very clear division between the front and back tiers. The *Client* tier does not worry about how to store the data and how to process mutiple requests, the *Server* tier does not need to take care of tricky user interface issues. Thus, it is a clean and modular design, this especially economizes application development time, and can also incure less network traffic and yield more security. Besides, because the business logic of the application can be separated from the user interface, this design leads to environment with homogeneous *Client*s, homogeneous applications and static business rules. Finally, as mentioned before, the technology has been available for a long time, it is therefore rather mature.

On the other hand, this architecture has obvious limitations. In terms of interoperability, since the *Client* tools and SQL middleware in two-tier environments tend to be propriety, in the case of a database *Server*, the stored procedures of the two-tier application are generally implemented directly using the commercial database management system's (DBMS) particular language. If they need to change or interoperate with some other types of DBMS, stored procedures need to be rewritten. In a two-tier design, the bulk of the application logic is on the *Client* side, each time the application version upgrades, the new version must be delivered, installed and

tested on each *Client*, the administration and maintenance become difficult. As for scalability, the two-tier design's performance capacity will be exceeded if it needs to serve more than about 100 users on a network(Software Engineering Institute (SEI), 1997c). This is because while the *Client* and *Server* exchange messages, the connection is kept alive even when no work is being done. The business logic in stored procedures is another limitation factor, as more application logic is moved to the database management *Server*, the requirements for processing power grows and then finally, both the network and the *Server* become saturated and no more capacity is left for further users.

2.1.4 Three-tier architecture (emerged in 1990s)

Three-tier architecture was developped in order to overcome the limitation of the two-tier architecture. The main difference is an additional layer which is located between the user interface (*Client*) and the data management (*Server*) components.

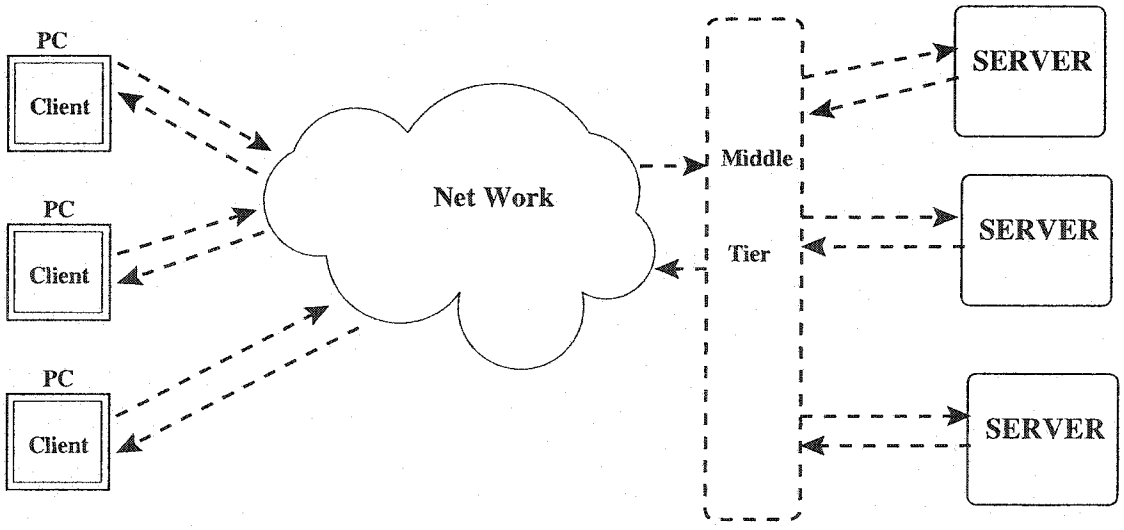


Figure 2.2 Three-tier architecture

1. Structure and functions:

This architecture allows for the separation of the business logic and data access. In Figure 2.2, the first part on the *Client* end, also called "presentation layer", has the same function as two-tier's *Client*; the third part, on the *Server* side, focuses on the database management functionality and is dedicated to data and file services that can be optimized without using any proprietary DBMS language; the middle part, also called "business logic layer", plays a very important role in improving the performance, flexibility, reusability and scalability of the total system. In this tier, developers can realize hightly optimized data indices, retrieval methods, and provide the replication, backup, redundancy and load-balancing procedures specific to the application needs. In addtion this tier can also add scheduling and prioritization of work among processes.

The middle tier centralizes the process logic, this separation of concerns makes database administration easier. All treatment of incoming requests are done in this part. If the developpers modify certain middle tier functions, the new version, which is placed on the middle tier *Server*, is available throughout the system and without impacting on the DBMS. This tier controls transactions and asynchronous queuing to ensure that requests from *Clients* are being processed in a resonable order; it also manages the cooperation of the distributed database, provides access to resources based on names instead of locations and thereby increases scalability and flexibility as system components are added or removed.

2. Usage Considerations:

The Structure of the three-tier architecture makes total applications easier to organize compared with the two-tier architecture. Since each tier can be built and executed on a separate platform, they can be developped in different languages.

The graphic user interface (GUI) can be both a heavy internet *Client* (written in a high level language such as C++ or Java) or a light internet *Client* (HTML, applet); C++, Smalltalk, Ada and Java can be used to build the middle tier(Software Engineering Institute (SEI), 1997b); and the third layer (database layer) can be programmed using mainly SQL.

To develop commercial and military distributed environments where many components share the resources such as heterogeneous databases and processing rules, three-tier architecture is a good choice. Another very important respect is legacy code reuse. With the deployment and support of a three-tier system, old database and process management rules can very well be maintained, and old and new applications can work side by side – this is the type of job that the VADOR project is aiming to implement.

3. Merits and shortcomings:

Three-tier architecture has been used successfully since the early 1990s on thousands of systems where distributed information computing in a heterogeneous environment is required. It is becoming more and more mature, and a common standard called "Open Software Foundation Distributed Computing Environment" (OSF/DCE)(The Open Group, 2000) offers a variety of features to support distributed application development.

At first, this architecture hides some of the complexity of underlying services (such as translating the requests into SQL, exchanging the data with the database system) and network communications. This flexibility increases performances especially in database manipulation, the *Client* does not have to understand SQL, and any modification to the organization of the database, such as names of tables or even the complete database structure does not need to be reflected back to the

Client part. Second, three-tier structure separates the functionality of services, and it allows for flexible developpment of individual tiers by application specialists. Finally, because the middle layer manages the complete processing of the *Clients*' requests, the whole system gets greater scalability and can thus supports hundreds of users.

Certainly, nothing is perfect, gaining the above advantages also incurs some costs and limitations. Because current development tools are relatively immature, the process of implementing three-tier architecture is still somewhat complex. A potential obstacle is how to promote code reuse and simplify maintenance of the code. Another potential problem is the separation of user interface logic, process management logic and data logic, which is not always obvious. Some process management may appear in all three tiers. Even though the developers choose to place some particular functionality in one tier, they should conform to some rules such as "ease of development and testing", "ease of administration", etc. There is no fail proof rule to always determine which tier should get which functionality, and increasing the number of tiers potentially increases the difficulty in determining which tier should implement a given functionality.

2.1.5 One-tier and N-tier architecture

Beyond two and three tier models, some experts presented the ideas of one and several-tier architectures (Java World, 2000) to complete the whole theory. Briefly, a one-tier application is simply a program that does not need to access the network while running. The apparent examples are desktop applications like word processor or compilers. One-tier architecture has a very big advantage: simplicity because it doesn't need to handle any network protocol on serialize data into streams. But, since it does not access the network, this type of architecture is not suited to

distributed programming.

N-tier architecture is based on component technology. The fundamental rule is: every applications should be designed to be universally reusable and accessible. Therefore, different applications should conform to a uniform interface and protocol. This kind of architecture has wonderful scalability, for example, in a stock trading system, multiple databases are accessed, multiple *Clients* are running various applications, and N-tier architecture then has overwhelming advantages. The price for the generalization is the speed. CORBA, which is the typical technology of N-tier, is very slow, it spends considerable time in marshaling and unmarshaling parameters, transmitting large amounts of data and handshaking the protocols (Java World, 2000). Besides, this technology is complex to understand and implement and yields higher product cost.

2.1.6 Technologies for Client/Server applications and Communications

The *C/S* model is very widely used in computer transactions, it is one of the central ideas of most business applications. One typical example is the World Wide Web (WWW), where the Web browser acts as a *Client* which requests the services (in its simplest form, getting a Web page) from Web *Servers* through the HyperText Transport Protocol(HTTP). Similarly, a computer with TCP/IP installed allows users to make *Client* requests for files using the File Transfer Protocol (FTP) *Server* from other computers on the internet. On top of these basic protocols, higher level technologies such as CORBA, from the OMG group (OMG group, 1997), Enterprise Java Bean (EJB), from SUN Corporation (Sun Coorperation, 2002) and Component Object Model (COM) from Microsoft (Microsoft Coorperation, 2000), are all based on the *C/S* model.

In this model, *Clients* and *Servers* typically communicate with each other using one of following methods: 1) Remote Procedure Call (RPC) where the *Client* process invokes a procedure which is remotely located on a *Server*, this procedure executes and sends the result back to *Client*. Since each request/response of an RPC is treated as a separate unit of work, every request must carry all information needed by the *Server* process. RPC is widely supported. 2) Remote Data Access (RDA) which allows *Client* programs or user-end tools to issue SQL queries to a remotely located database. RDA is heavily supported by database vendors. 3) Queued Message Processing (QMP) where the *Client* messages are stored in queues and the *Server* works on them when it becomes free.This method allows the *Client* to asynchronously send requests to the *Server* and the request to be processed even if the sender is disconnected from the *Server*.

2.1.7 Remaining challenges

In the process of developping a distributed system which is mainly based on the *C/S* model, there are still many issues that developers should consider in order to take full advantage of the potential offered by this technology.

A first issue is concurrency control. The resources are always shared by several *Clients* in a distributed system, therefore, it is possible that many *Clients* may access the same data at the same time. On the *Server* side, this means the *Server's* operations performed on behalf of different *Clients* may sometimes interfere with each other. Basically, concurrency control can be achieved either by *Clients'* transactions waiting for one another or by restarting transactions after conflicts have been detected, sometimes the resolution combines the above two.

Another important issue is security. If a computer system handles some highly sen-

sitive information such as financial transactions or confidential, classified messages whose secrecy and integrity are critical, it is very likely that this computer will become a potential targets for malicious or mischievous attacks. In networking, the security issue includes two challenges: 1. How to send sensitive information in a secure manner, 2. How to identify a remote user or other agent correctly. Encryption is a common method to tighten security, it is a process of encoding a message in a certain way to hide its content. Modern crytography includes several secure algorithms for encrypting and decrypting messages. Digital certificates are another security method which is used to find a proper remote user to contact; it is a document containing a statement signed by a principle. Firewalls and credentials are used to check the illegal access to a shared resource.

Applications using the *C/S* model should also consider scalability, so that the system remains effective when the number of shared resources and users are increased. Handling of failures when faults occur in either hardware or software, and coping with heterogeneous environments which consists of different operating systems and different programming language are also issues related to scalability.

2.2 New world of mobile agents

Traditionally, most distributed applications have been based on the *C/S* paradigm, but in this model, as mentioned above, communications only happen between objects whose code is statically resident on specific hosts. To remove this restriction, a new concept called "mobile code" is introduced.

2.2.1 Mobile Agents technology

A runtime system allows the application to interface with the environment in which the application is running. In this kind of system, a software module can be dynamically run on a host where that module is not statically installed and configured. This allows the code to possibly "move" from a node to an other in the network.

A mobile agent(*MA*) is defined as a self-contained software element that acts autonomously on behalf of a user. Each agent has its own thread of execution which means it can perform tasks on its own initiative. In addition, a *MA* has capability of migrating under its own control from machine to machine in a heterogeneous network. When an agent wants to move, it can suspend its execution at an arbitrary point, transport itself to another computer and then resume its execution on the new computer from the point where it left off.

To develop a mobile agent system, at least two key issues should be considered. The first one is "autonomy". This feature allows the agent to act on its own by using the data and the mobile logic without any human intervention or guidance. After an agent is sent, the user can undertake some other jobs while waiting for the final response of the task. The time needed for the task execution is also reduced because interaction between the user and agent is avoided. Agents should be designed to deal with any condition that may occur during execution. In Multi-agent systems, an agent which comes from a remote machine is called "external agent". Its most important duty is to fetch the necessary information from the destination host. Thus, an autonomous agent should be able to contact the platform and the "local agents" (the agents that are already running on the platform). This functionality can be implemented through CC (Communication channels), which are the minimal features of a *MA* platform. After an "external" agent gets on a

machine and is verified its identification, a trust relationship should be established between this agent and the "local agents" or the platform so that they can exchange informations through CC and cooperate each other. Due to the fact that an agent can collect, manipulate, and distribute data (sometimes private data), it is possible that security problems might occur when the agent communicates or exchanges data with its environment or other agents belonging to different systems. This will be discussed further in section 2.2.3.

The second key issue refers to the mobility of an agent. Mobility enables the agent to travel to a host where the needed data is physically stored. Since transferring large amounts of data always causes high network traffic, a good solution to accessing a remote database is to migrate an agent. In the VADOR project, this idea will be examined to determine whether agents can be used for utilizing the capacity of distributed machines reasonably. The agent accesses the resources locally and brings back only the result to user, this approach causes low network traffic and a better use of network resources. The migration of the agent should respects the rules concerning the agent transport, agent interaction and security. Section 2.2.3 will discuss standards in the *MA* world.

There are still some technical issues that should be considered while developing a mobile agent-based system: agents' mobility, security, and standardizing of the communication methods. These issues are discussed next.

2.2.2 Code mobility issue: strong or weak

The *Mobile Agents* approach has already been described as a good way to easily design and maintain distributed systems. However, the use of *MA* does not come without a cost. Code migration incurs additional network traffic, resource con-

sumption of the current host, and time delays. Several kinds of mobility have been developed, and it is a challenge to balance the benefits and disadvantages of the different types of mobilities within the scope of a project. The attributes of each type of mobility are listed as follows:

1. Constrained mobility:

This is the most elementary form of code mobility. The *Client* can dynamically enhance the *Server's* capability by sending code to the *Server*. At code arrival, a *Client* will remotely initiate the execution of the code. Hence, the *Client* owns the code needed to perform a service that can access the target host's resources. By giving different requests, the code can be made to implement various services.

In fact, this type of approach can be seen as an extension of the *C/S* paradigm. The migration is only from the host of the *Client* to the *Server*, the agent does not need to be very sophisticated or particularly autonomous, because the developer provides almost all the information for the future steps of the execution. It also does not need to incorporate complex migration features and its size is light enough to keep network traffic low. In addtion, it doesn't require a general *MA* platform. However, since the code which is sent to the *Server* host is not sufficiently autonomous, the *Client* program should interact a lot with the code to guide its execution.

2. Weak mobility:

This is the ability of a mobile code system to allow code transfer across different hosts, it's also called "non-Transparent Migration". The code may be accompanied by some initialization data, but no migration of execution state is involved. In this case, the code can be migrated either as stand-alone code or as a code fragment. The stand-alone code is self-contained and will be used to run a newly created execution unit on the destination site. A code fragment must be linked in the

context of already running code and eventually executed (Fuggetta, 1998).

Weak mobility does not retain any knowledge of the data processed or the actions performed in previously visited hosts. It is in contrast with standard definitions of *MA*, because one of *MA*'s main features is automatic resumption of threads' execution. Mechanisms supporting this approach can be either synchronous or asynchronous. In the asynchronous case, the actual execution of the code may take place in an immediate style, which means the code is executed as soon as it is received, or in a deferred style where the execution is performed only when a given condition is satisfied.

3. Strong mobility:

This ability, also called "Transparent Migration", enables an execution unit to move as a whole by carrying its execution state (such as instruction pointer) across migration. It is trully transparent to migration so that the executing unit resumes execution on the new host right after the instruction that triggered the migration. During the period of travel, the agent can accumulate informations it collects, and preserve them until finishing the total trip. This specifically allows for the implementation of tasks in which the agent operations depend on data gathered in previous hosts.

Strong mobility needs a "remote clone" (an agent totally copies itself into two agents, and these two continue their execution from the same point but work with different data and at different places), and "suspend" mechanisms (an agent supends its execution, saving current state for the future resumption). The migration can be either "proactive" or "reactive". The former means the time and destination of migration are determined autonomously by the migrating agent itself, whereas the latter means that the movement is triggered by a different executing unit that

has some relationship with this agent.

Some experts presented another type of mobility based on strong mobility, it's called "full mobility" which implies the movement of the whole state including all threads' stacks, name-spaces and other relevant resources (Bettini, 2001). Certainly, this style of mobility makes the migration more general and completely transparent, however, it requires a very strong support from the operating system layer.

4. Comparision and applications:

As was discussed above, constrained mobility's advantages are light-weight and appropriateness to a specific task, but its largest shortcoming is its lack of autonomy – one of the most important features of *MA* technology. This makes this style of mobility look more like a *C/S* model rather than *MA*.

Weak mobility realizes true autonomy, it is also relatively easy to implement since it does not need the program to take care of many executing states. Due to this characteristic, no pointers or stack status need to be manipulated. Therefore, this approach is less dangerous because it is less likely to cause memory lapse.

Strong mobility displays the complete set at features expected from *MA*, but the mechanisms which support this approach must pay the price for more complex development and maintenance. One challenge is how to get the current execution state and keep this information in the agent. The other challenge is how to recover this information on the target host. The execution state includes many complex content, for example, the executing stack in memory. If an agent needs the help of some classes that are not on the target host, it also must carry the code of those classes to its destination or utilize RPC to get the necessary service. Besides, the transaction of the process's state is an additional work load for the network.

Furthermore, while one of this kind of agent is migrating on the network, because it must carry every valuable information it gathered, the increasing backpack will absorb large amount of network resources. There must be a limit on the number of nodes it can visit (generally, the number is less than 100 but depends on the size of every message).

Different mobilities have different utilization fields. Normally, weak mobility can fit applications of "running a task". For example, in the VADOR project, each agent carries a series of missions and walks through every possible node to complete all the tasks. In this case, it is not necessary to record the states of every execution, weak mobility is enough. Another example is a stock deal-trigger which can monitor the proper price of some specified stocks and can make a deal at proper time. Strong mobility is particularly well suited to data-intensive tasks which involve data aggregation and on-line data analysis. A simple example is a web site search engine which can collect the information according to user's preferences.

5. Current projects on code mobility

Many projects have been developed that implement mobile code technology, some of them support weak mobility whereas others support strong mobility. Here is a brief survey of current research project on *MA* and their chosen mobility level.

Java Aglets (Lange, 1998) is developed by IBM Tokyo Research Laboratory in Japan. It supports weak mobility. This project provides two migration primitives, one is "dispatch" which performs code shipping of stand-alone code to the context specified as a parameter, its mechanism is asynchronous and immediate. The other is "retract" which is used to force an aglet to come back to the context where retract is executed, the mechanism is synchronous and immediate.

Agent Tcl (Gray, 1997) is developed at the university of Darthmouth. This project

has an interpreter extended with support for strong mobility. It's an extension of the Tcl language which adds an explicit statck to the Tcl core so that it can capture the complete internal state of an executing unit (agent).

Moreover, Ara (Holger Peine, 1997) from University of Kaiserslautern and Sumatra (Vitek, 1997) from University of Maryland, also both support strong mobility. Meanwhile, Grasshopper (IKV++ Technologies AG, 2002) from IKV++ Technologies, Mole (Baumann *et al.*, 1997) from University of Stuttgart, Concordia from Mitsubishi Electric Information Technology Center America, and Voyager from ObjectSpace company (Recursion Software Inc., 2001) support weak mobility.

2.2.3 Security issues

Security is another critical issue which should be considered. *MA* systems make arbitrary code roam and execute on remote host and, sometimes, this behavior can cause security threats. The platform may steal private messages carried by an agent, or tamper with the agent's content so that it can be controled to attack target hosts. A malicious agent might try to access or even destroy privileged information in the current host, or consume all resources. And since the network environment is open, other attacks on both agents and platforms might come from all coners of the world. A mature *MA* system should be able to detect and prevent these threats.

1. Analysis of the threats

Malicious attacks may come from different angles, and hurt various respects of the computing environment. The following paragraphs summarize these attacks according to the assailants and victims (Wayne Jansen, 1999).

Attacks from agent to *MA* system: a rogue agent can launch denial of service attacks by running harmful scripts to disrupt the host's system, or degrade the performance of the platform. If the *MA* system doesn't have good enough access control mechanisms, unauthorized agents may have a chance to access the service and resources which they are not entitled to. Another method of attacking is to masquerade, which means that an unauthorized agent claims the identity of a legitimate agent to pass the authorizing examination.

Attacks from agent to agent: in a system which allows agents to communicate with each other directly, a malicious agent may attempt to cheat other agents to get some important information such as credit card numbers or account's passwords. An other method of attack is to repeatly send messages to the victims until they can't carry the burden any more and collapse. Sometimes, the agent platform components are also agents themselves, this type of attack may then shut down the total agent service on a host. Besides, bad agents might also illegally access sensitive messages of other agents which have weak control mechanisms.

Attacks from *MA* system to agent: a malicious *MA* platform can masquerade as another platform to deceive a mobile agent; if the poor victim gets on this host, some crimes such as pulling out private information, modifying content or even shutting down the agent will be conducted. Rogue platforms can also eavesdrop the communication between two agents to get the information in which it is interested, the results of eavesdropping are unpredictable, from sending many unsolicited advertisements to stealing the users' money or hurting the users' reputation.

2. Requirements of MA security issue

To resist all these kinds of attacks, at least four elements are required in the security part of a *MA* system: authentication, confidentiality, integrity and sandboxing.

Authentication mechanism is used to identify the subjects and objects representing the participating entities. It is the fundation of the other security mechanisms. To be safer, each agent on the *MA* platform should take responsibility for its actions, therefore, it is necessary to identify, audit, and then authenticate the agents (user's feature) at the entrance of the platform, or at the begining of a communication with another agent. Of course, the platform also must be able to authenticate its identity to incoming agents or other platforms where the local *MA* wants to go.

Confidentiality mechanisms are used to protect the data carried by agents from being stolen. On the itinerary of an agent, the confidentiality of its data can be achieved at each host by a simple enciphering. Up to now, RSA-based encrytion (RSA security, 2002) and sliding encryption (Young, 1997) are vailable for *MA* systems. In some cases, *MA* may also want to keep their location confidential, platforms should give the agents a chance to decide if their presence will be made publicly available or not, and enforce different security policies depending on the repective choices.

Integrity is for protecting another aspect of the agent's data. An agent performs a function on behalf of the user, for example, to collect price information from different retailors. One issue users are concerned with is whether the data has been changed illegally. Attacks against agents realize their goals by changing an agents' destination, resuming an old message or even replacing an entire information. The agent itself can not prevent such malicious attacks, but it can use integrity mechanisms to take measures to detect the tampering.

The sandboxing concept is introduced to prevent an agent from damaging the platform in the course of its execution. It limits the code running in a very restricted environment so that an untrusted agent can be executed without any worry. Certainly, this mechanism also limits the usefulness of agents.

2.2.4 Standardization issue

A mobile agent is always seen as a member of a community which contains a group of heterogeneous software processes. They are designed to interact with each other so that they can work cooperatively to achieve mutual goals in various fiels such as electronic commerce, network management, real-world situation control, etc. To realize such an objective, a very critical step is to implement interoperability among agents developed by various enterprises. Therefore, a common specification about the agents' interaction is essential, that allows developer to look up the standard communication protocol and build common agent processes according to the agreed architecture.

After several years of international effort, some standardization criteria have been established. They target either the communication mechanism (such as CORBA, RMI) or the communication language between agent and platform or two agents. The most common standards are the "Foundation for Intelligent Physical Agents" (FIPA), the "Knowledge Query and Manipulation Language" (KQML), and the "Mobile Agents System Interoperability Facility" (MASIF)(OMG group, 1998) standards. They will be discussed in the following sections.

1. The present situation of Agent Communication Language (ACL):

Since agents need to be able to interact with other agents or their environment, it is necessary to set up a inter-translatable representation language among them. A good ACL should have some critical features: its form should be concise, declarative and readable by people; its content must be clear and precise enough to express communication acts or agent information, and also should fit well with other system; the implementation should be efficient with respect both to network's speed and bandwidth; besides, it must make the communication reliable

and secure (Stanford, 1997).

So far, there is no globally agreed upon ACL available, however, standards such as FIPA ACL and KQML have emerged as potential candidates in this respect.

2. FIPA and FIPA ACL

FIPA(The Fundation of Intelligent Physical Agent, 2000) was formed in 1996 as a non-profit organization by OMG. Its primary purpose is to create consensus between different agent technology organizations. Its other goal is to specify only the system components' external behaviors, and leave implementation details and internal architectures to developers. FIPA targets the commercial field rather than academic field, many organizations from real markets have shown their interests in it. Currently, it has at least 40 international member organizations such as IBM, Alcatel, HP, etc.

In 1997, 1998 and 2000, three versions of FIPA specifications were introduced, they defined the communication criteria for different aspects: "Agent communication" is in charge of interaction between agents; "Agent management" supports the location and creation of agents; "Agent/software integration" provides interaction between agents and other non-agent software; "Human/agent interaction" gives the users a chance to manipulate the agents and the platforms.

FIPA ACL is defined in the "Agent communication" section, it facilitates communications between agents and could support agents interactions such as negotiation, co-operation and information exchange. It can be viewed as consisting of the following 5 elements: 1) the "protocol" offers the social language for building dialog between agents, 2) the "communicative act" expresses the request's intention, 3) the "messaging" is used to help the postal behaviors such as identity of sending or receiving agents, 4) the "content language" defines the sematics and grammar of

the letters among agents, 5) the "ontology" sets up the vocabulary and meaning of the terms and concepts used in the letter's content.

3. Knowledge of KQML

Similar to FIPA ACL, KQML(Tim Finin, 1994) is another effort to standardize the communication language between agents. It was developed by a consortium called ARPA- Knowledge Sharing Effort (KSE) whose goal is to develop conventions facilitating sharing and reuse of knowledge bases and knowledge-based technology.

KQML uses a "wrapper" mechanism to treat the agents' information, it is conceptually a layered language. Every time the agents use KQML, the message which is composed in various representation language is wrapped in a KQML message. There are three layers in KQML: the core layer is called "Content layer" which encapsulates the message. In fact, the implementations of KQML does not care about the message's format or grammar, the content layer ignores those differences. The middle layer is the "Message layer", it defines the interactions between two KQML-speaking agents, its main task is to identify the protocol to be used to deliver the message and to supply a speech act that the sender attaches to the content. The outside layer is the "Communication layer" which provides some service for the communication such as identifying the sender and recipient.

KQML's communication architecture mainly includes two components, the "Router" and the "Facilitator". Every agent has an associated Router, the "Router" handles all KQML messages which are sent or received by the host agent. In fact, the "Router" acts as the only contact point between the agent and its outside environment. The facilitator is used to provide information services and treat the requirements of other agents, its services comprise content based routing, brokering or smart multicasting of information, and suppliers recruiting. It is actually an

agent, each "Facilitator" manages one local group of agents.

4. Choosing: FIPA ACL or KQML

In many respects, FIPA ACL is very similiar to KQML, they are both independent of the content language, use ontology and messages. But their differences in functionality and semantic framework leave both of them with merits and shortcomings.

Compared to FIPA ACL, KQML was developed earlier. It is eight years old which means that it has been experimented with, and developers have more experience on using it. It is also used more widely in the agent community. KQML provides the "Facilitator" to overcome the different content languages used by different systems and offers a mach-making service. However, KQML's disadvantages are also obvious. First, it has no formal specification sanctioned by some consensus-creating body, this partly explains why some messages still have an ambiguous meaning. Also, it has no agreed-upon semantics foundation, and in fact, KQML allows the developers to build their own semantics at will. As a consequence, agent system developers have invented many dialects, and interoperability has become weaker and weaker among agents even though they are all KQML compliant agents.

FIPA ACL has remedied those KQML's weaknesses; it has been produced by a well-organized standard body with a concrete agenda and an excellent commitment to agent system interoperability. Further developments will give it more vitality, and attract more attention from commercial and academic areas.

5. Introduction of MASIF

In order to promote the interaction among different *MA* platforms, MASIF was developped through the cooperation of several persons and organizations, and was

finally adopted by the Object Management Group (OMG) as a standard. The idea of MASIF is to achieve a certain degree of interoperability between *MA* platforms which are produced by different manufacturers without enforcing thorough modifications of the programs.

MASIF has standardized some important *MA* operations, the "management" provides the agent systems with the opportunity of manipulating the agents from outside, it allows the agents' creation, termination and suspension, and the resuming of their execution; the "transaction" defines the necessary operations for that agents to migrate among different systems; the "identification" is in charge of identifying the *MA* system entities such as agents, *MA* systems and places, these entities all have the identification features like "authority", "identity" and "agent system type".

Different from FIPA and KQML, MASIF's primary focus is on mobility issues, it focuses on the low level communication methods while agents are migrating from one platform to the other. The MASIF effort can be regarded as a "bottom-up" activity whereas FIPA is more "top-down". OMG's idea is to integrate these two standards together to get the benifits of both. So far, some platforms have already been developed that realizes this goal, for example, Grasshopper is the first FIPA and MASIF compliant agent platform (IKV++ Technologies AG, 2002).

2.3 Mobile Agents model versus Client/Server model

As mentioned before, people have used the *C/S* model for a very long time and gotten plenty of experience with it. Today, *C/S* approach is the most common way of implementing distributed applications. This lengthy use of the model has revealed its shortcomings, and people have developped several techniques to en-

hance the model. Typical more recent techniques include: Remote Procedure Call (RPC), message passing (MP), Remote EValuation (REV), applets and process migration (PM). But each technique focuses on one problem and it is not easy to remain compatible with others. Developers usually have a hard time in combining these techniques together so that they extract full benefits of each. Very different from *C/S* model, mobile agent technology can cover and unite most of the *C/S* model's advantages. The following sections present both the *MA*'s advantages and disadvantages when compared with the *C/S* model.

2.3.1 Comparasion between the two models

When faced with the problem of dealing with highly distributed processes, the *C/S* model starts to break down. In order to perform adequately, the *C/S* model needs reliable connections and good bandwidth since data must be copied between the nodes on the network. Moreover, the traditional *C/S* model provides fixed *Server* side operations, the requests from the *Client* side may not be answered properly through a single communication step. These characteristics of the *C/S* model have a direct impact on the performance when the quality of the network connection is poor, or to maintain applications if their functionality is constantly changed.

The *MA* paradigm overcomes these problems since it does not use the request/response architecture. It moves the flow of control across the network, and transforms remote treatment into a local treatment. Once the agent arrives at the target host, the *Client* side does not need to send further requests to guide the agent process. This partly explains is why the *MA* paradigm is often considered more robust than the *C/S* model. In addition, *MA*'s scalability is also higher, because in a *MA* system, the *Server* administration's responsibility is simply to manage systems and to monitor local load. The whole system is solely composed of agents, the relationship

between the users and *Servers* is thus totally coded into agents as opposed to the *C/S* paradigm. In the long run standardization which is completed by the organizations such as OMG, there will be no application-level protocol used by *MA*, every agent and platform should conform to the same communication language, and therefore, the *MA* paradigm will enforce collective agent compatibility.

Let's now compare one by one some modern *C/S* techniques with the *MA* paradigm to see the detailed differences.

1. MA vs MP and RPC

MP is an earlier communication method which transfers messages via files or pipes. The application at the *Client* end prepares a message into a tagged or structured text file, and sends it to the target location through a message transport service. Since this mechanism is asynchronous, the response from the *Server* side is not immedately available, and the *Client* side application needs to store the state and recover it when the response comes. This characteristic makes MP very effective for "one to many" communications, but for the "one to one" model, MP has less throughput than some other methods (Brewington, 1999).

RPC is used to simplify the request-response process in the *C/S* model, it allows a *Client* program to invoke a *Server* side procedure. The communication is based on a pair of stubs on both sides, *Client* and *Server* only contact with the local stub for exchanging their information, and at the lower level, those stubs manage the data transactions. The main advantage of this method is high efficiency and low latency in "one to one" communication. The principal shortcoming of the method is that while using the synchronous RPC, which is the traditional one, the *Client* must be blocked until the *Server* gives back the result. Although there are some extensions to allow launching concurrent invocations of the same procedure

on different *Servers*, applications are difficult to realize. In addition, this technique is not easy to use when incremental results must be sent from *Server* to *Client*.

MP and RPC have a fatal weaknesses in terms of network resource consumption. Since the *Client* is limited to the services of the *Server* side, and it is unlikely that the sever's operations will exactly suit the *Client's* requests, the *Client* has to make several contacts with the *Server* to get the final result, thus ensuring many inevitable transactions of intermediate data to happen.

In contrast to the above two *C/S* methods, *MA* can migrate to the location of the *Server*, and invoke many necessary operations without transferring any intermediate data across the network. *MA* can thus conserve bandwidth and therefore reduce latency. In addition, *MA* can keep working even when a temporary interruption of network happened between the *Client* and the *Server*. As long as *MA* can start to work on a new host, it will not need the link any more until it is ready to send back the result, its autonomy can make sure it can send the message at proper time.

2. MA vs REV and applets

REV is used to fix one of RPC's problems – unnecessary data transaction. One piece of the *Client* side application can be sent to the *Server* side and be executed on behalf of the *Client*. REV also uses *Client* and *Server* stubs to send procedures or data, it is in fact a variant of the RPC. The main limitation is that the procedure must be self-contained, and every necessary external function or variable must be provided on the *Server* end or travel along with the procedure.

Applets are another technology which supports moving a piece of code to a remote location. For example, when people visit a Web site, applet programs become available to be downloaded onto the user's machine and perform some function.

We can see that both of these two technologies demonstrate some mobility features, but *MA* is more flexible than both of them. First, *MA* can migrate to any other possible hosts at will. It decides on the next station according to its needs, whereas REV and applets can only move to previously determined places. Second, *MA*'s migration can be from *Client* to *Server* or vice versa, but REV and applet can only realize one-way migration. Finally, *MA* can migrate as many times as desired.

3. MA vs Process Migration

Process Migration (PM) is the act of transferring a process between two machines during its execution. When the migration happens, the process code, data and some critical states such as the process's address space, execution point and communication state will be transferred to a new host, and then relaunched as a new remote process.

PM is closer to *MA* than most other methods, but there are still apparent differences between them. The main difference is that PM systems do not allow the process to choose when and where the process should migrate, whereas *MA* has enough autonomy to implement these functions. Also a PM system is always considered to work in a closed and clean environment where security problems are less of a concern, but *MA* is based both on platform - independent and secure in open environments paradigms.

2.3.2 Arguments on *MA*'s strengths in applications

After the above comparasions, we can see that *MA* shows many advantages in various fields. But it doesn't mean that *MA* is always better than *C/S* under all conditions. In fact, sometimes, *MA*'s advantages are relatively slender, and usually, equivalent solutions can be found under *C/S* model. The following parts

will present several applications where both *MA* and *C/S* can work, and therefore compare their effects.

1. Transaction of requests and responses

From the previous discussion, *MA* seems to have an overwhelming advantages over *C/S* in transferring messages since the interaction is done locally. But under other respects, such as recovery, their capabilities are not as obviously different. The recovery property means that if a *Server* becomes unavailable to a *Client*, the *Client* application has the ability to redirect its request to another or even to several other *Servers* to get the same final result.

In this respect, *MA* should clearly have access to a method providing information about alternative *Servers*. That information is not supposed to be carried with the agent since it is likely to be out of date as the time passes. The agent must then check back with the dispatching *Client* which maintains the knowledge needed to refresh itself, or the agent must know of a fixed place where it can obtain the information. RPC maybe a little more robust in this respect, if one RPC call fails, the *Client* applications will check the local database searching for a new target host. RPC can also verify alteratives with the user.

2. Maintenance of execution state

If the requests from the *Client* include series of complex operations that will spend a lot of time or need the cooperation of several *Servers*, keeping the intermediate execution state becomes essential. In the *C/S* paradigm, applications on both sides should provide the mechanism for making the process state persistent. It is a heavy burden on application development. For *MA*, since each agent carries its own state during the migrations, the sending computer is relieved from the duty of state preservation. But on the other hand, this mechanism may enlarge the agent's

interaction with the target *Server* because it must describe much more about its context. This also results in some useless data transaction while an agent is moving. Nevertheless, the *MA* based operation is still more recoverable than *C/S* operations such as a RPC based operation, because *MA*'s mechanism is asynchronous. There are no strong time constraints for saving the state, and the state is saved at the application level rather than at the process level, therefore, detecting and recovering the agent's state becomes a relatively easier job when restarting a lost agent.

3. Semantic information retrieval system

Semantic information retrieval systems aim to support remote information searching, filtering and retreival. When a user request is entered on the *Client* side, the system interprets this query semantically, and sends a reformulated query to one or more *Servers*. Once information have been retreived, the *Server* then informs the *Client* side of the final result. The main issue here has to do with the interaction between the mutiple *Servers*, because the target *Servers* may be widely distributed over the network and the resulting information may be huge.

A more efficient approach is to send the user's perferences to the data source rather than send the data back to the *Client* side program. The process of searching and filtering or even the summarizing should be accomplished locally at the data source. *MA* has some apparent advantages in this respect since it gets closer to the actual data source, it can not only do automatic identifying and indexing of a small number of interesting documents from a large number of uninteresting ones, but it can also contact other local agents to share useful results. In the *C/S* paradigm, the equivalent solution is to install a standard retrieval and filtering applications (search engines) at every information repository. The requests of the *Client* side can launch the search engines on each target host. This solution must confront the difficulty of agreeing on a standard search engine or search protocol.

4. Supporting mobile devices

Low capacity mobile devices such as third generation mobile phones are different from static machines because of their inherent mobility which accomodates people working in varying locations. The price of this characteristic is the difficulty of connecting to a network and the assocoated poor capacity. Generally, mobile devices don't keep a steady connection to the network, hence have only intermittent access to *Servers*. Because most of them use dial-up lines, the bandwidth of the connections are usually relatively low. Besides, their storage and processing capacity is sometimes relatively limited.

In this case, *MA*'s advantages are fully expressed because agents always work locally with the *Server* and allow for intermittent connections. A mobile device can launch agents during brief connection sessions, and get the result in the next short online period. As mentioned above, agents perform the information retrieval and filtering at the *Server*'s location, useless data transactions are ignored, and futher more, the result data size is minimized to save the capacity of the mobile device. The *C/S* applications have relatively weak capabilties in supporting mobile devices' operational constraints, their main advantage is that they can transmit only some queries rather than a chunk of mobile code across the network, this can reduce the consumption of bandwidth, but can not keep working during disconnection.

2.3.3 Summary

Based upon the above description, we can tell that the *C/S* paradigm can provide alternative methods to replace the *MA*, however, if we look at the total advantages that *MA* has, we know *MA* is still a stronger way. In general, we can say that the *MA* model offers the following merits over the traditional *C/S* model: first, it

economizes the bandwidth by filtering out irrelevant data before results are finally generated; second, it can autonomously migrate to other machines according to the necessity; third, it can efficiently work because it can move closer to data; forth, ongoing processing doesn't need continuous network connection, it works more reliably; finally, contain more scalability in the highly distributed system.

In order to practice these *MA*'s advantages, in the next section, the utilization of *MA* technology in VADOR project will be presented.

CHAPTER 3

DISTRIBUTED EXECUTION OF LEGACY APPLICATIONS

Based on the previous discussion and literature survey, two approaches to the implementation of a *CPU Server* are described. The first is based on the *C/S* model, while the second relies on the *MA* paradigm. In the case of the *CPU Server* part of the VADOR project, both models can achieve the same propose, the main differences arising from two aspects: efficiency and ease of use. This chapter first presents the responsibilities of the *CPU Server*, and then discuss the details of two solutions, their respectives advantages and disadvantages may be compared.

3.1 Overview of CPU server's behaviour and functionality:

Expressed simply, remotely executing a legacy code involves launching an external application command from a program, passing in some necessary parameters which indicate the paths of input and output data files, and other approriate informations in string format. The description of the whole procedure is not as simple as it looks, though, since the *CPU Server* must manage transactions on all data files, and process required internal synchronisation and sequencing commands.

3.1.1 Input and Output data files:

To the *CPU Server*, the most complex task is dealing with data file attributes. Data files are classified into two categories: input files and output files.

The input data files must be physically existing before the legacy application is ready to run, but sometimes, they are not located in the host where the *CPU Server* is running. The VADOR project allows these data files to be located on any reachable machine. In cases where there is no NFS mounted disk to allow simple access to the file, the *CPU Server* will transfer the files to its local machine. Therefore, a temporary directory should be set up to contain the incoming files. The *CPU Server* has a function that can check whether or not the input data files are ready at the local host. This function is ignored in cases where the input files are tagged as "optional".

At the end of the execution process, resulting output data files are created after the application have been run. They also contain the attributes of "local" or "remote". To determine this attribute, the *CPU Server* needs the host names of the files to accompany the files' paths. Because output data files can not be generated directly on the remote host, the *CPU Server* must first create those files in a temporary local directory, and then make some remote transactions to send the files to their final destination.

3.1.2 File transaction method

The data files from remote hosts need to be transferred. A common method of transaction is to use a pair of sockets. A data stream should be built between the sockets on the *Client* and those on the *Server*. When all the data reaches the target machine, sockets on both sides should be closed. This simple method can cause some security problems, for example, because the socket program has permissions of a common user at the *Server* side, the user at the *Client* side may access some files which are not supposed to be exposed to the outside. Therefore, within the VADOR framework, the *CPU Server* utilizes the *Apache Server* to transfer the files

instead of the socket method.

The *Apache Server*(The Apache Software Foundation, 1996) is the most popular web *Server* on the Internet; it is an effort to develop and maintain an open-source HTTP *Server* for various modern desktop and *Server* operating systems. It is a secure, efficient and extensible *Server* which provides HTTP services fully compatible with the current HTTP standards. The VADOR project assumes that within each NFS domain, there is one *Apache Server* running, so that if the *CPU Server* needs to make a file transaction, it will build an "input-stream" with the remote *Apache Server*, and download the necessary files. The benefits of using the *Apache Server* are numerous. First, the *Apache Server* provides mechanisms to protect the *Server* side files. It restricts and manages the access permissions, the *Client* side can only access files in the directories which are configured in the *Apache Server's* configuration file. The *Apache Server* can also set the range of requests sources, if a request comes from an address that is not in the "source addresses list", the *Apache Server* will refuse it. The *Apache Server* also realizes the Secure Sockets Layer (SSL) protocol, thus providing high degree of confidentiality while transfering sensitive data files. An other advantage of this solution is that since file transactions are based on a standard approach, in the future, when the *Apache Server* gets improved, the *CPU Server* can immediately benefit from these enhancements without any code modification.

However, there are still two issues which should be considered carefully. The first is: on the remote host, the absolute path of the target file is different from the URL address, and the *Client* side must be instructed on how to translate between these two sets of locations, a second question then becomes how does the *Client* convert the file path into the right URL address string for downloading? The other issue is: file transaction includes not only downloading (getting the input files from

the remote machine), but also uploading (sending the output files to the remote machine), the question then becomes how can the *Client* send files back to a remote *Server*?

Solution to both of these two issues involve the use of a Java application running on the *Server* side. If a downloading request is issued, the java application can translate the file's absolute path to the URL address, and feed it back to the *Client* side. In the case of uploading, the Java program will be used to launch a java thread to perform a download operation from the *Client* end, in other words, uploading is thus translated into a download from the other end. Basically, the *Apache Server* can not invoke a java application, it needs the help of the *Tomcat Server*.

Tomcat is a servlet container that is used in the official reference implementation for the Java Servlet and JavaServer Pages technologies. It is developed in an open and participatory environment and released under the Apache Software License. Through proper configuration, the *Tomcat and Apache Servers* can work together. In the VADOR framework, when a request comes, Apache and Tomcat can launch the appropriate java servlet program to realize the translation and uploading tasks.

3.1.3 Internal Commands Execution

Apart from running legacy applications, the *CPU Server* needs to accomplish some other commands as well, which are defined in the VADOR project. Sometimes, VADOR needs the *CPU Server* to perform certain operations pertinent to file management, including copying or moving a file, getting the list of file names under a directory, removing some files, changing the attribute of certain files, etc. The *CPU Server* also has to deal with the internal commands that come from the *Executive Server*, such as "stop task", "pause task", "resume task", so that

the *Executive Server* is able to control the status of a task. Moreover, the *CPU Server* needs to manage the requests about its own threads that are in charge of file transactions. These requests are divided into two categories: "register thread" and "log out thread." According to executing duration, there are two different types of internal commands: synchronous and asynchronous. If the command is synchronous, the *Executive Server* will keep waiting until the result is fed back since this kind of command is expected to execute in a very short time. This approach conceptually uses an "always connected" C/S model (see the section 2.1.3). In the case of an asynchronous command, since the *CPU Server* cannot finish the execution immediately, both the *Executive Server* and the *CPU Server* must keep a task ID number of this command in order to be able to identify it when the result will be returned.

3.2 The Client/Server implementation of the CPU server

Let us first examine the complete architecture, behaviour and encountered issues when using the *C/S* model to implement the *CPU Server*. The key communication method of this solution is to use a socket stream. Both the *Executive Server* and the *CPU Server* listen to an agreed port. If a message (for example, the content of a new command or a result of an execution) needs to be transferred, a stream between the two *Servers* is established. Once the transaction is finished, the stream is disconnected, and the computer resources are released. In the *C/S* structure, the critical issue becomes how to transport the execution information efficiently and accurately.

3.2.1 The challenge of sending task information

Each time the *Executive Server* asks the *CPU Server* to accomplish a task, it always passes information that describes the entire profile of the task. Sometimes, a task only triggers a very simple and fast operation, for example, remove a file. Some tasks, on the other hand, may become very complex. In some cases, it is difficult to have a task done in only one single step. For instance, the *Executive Server* may need to run a legacy code, and that legacy code may require some prerequisites, such as preparing a suitable directory, setting up some pertinent environment variables, etc. After the execution, the task is still responsible for making backup copies of the output files, or changing their group names to make them accessible. Such a task may contain several sub-tasks, some of them are internally defined commands, whereas other are external ones; their execution must follow a certain order, some sub-tasks' results are the preconditions of the others' executions. Furthermore, the time needed to execute each sub-task is different, lengthy and short commands are mixed together. Some special tasks are even sent to be queued on super computers to run.

To coordinate these complex tasks, one solution is to let the *Executive Server* divide the whole task into small parts. Each time the *Executive Server* contacts the *CPU Server*, only one part of the task or a group of sub parts to run in parallel would be transferred. The connection between the two *Servers* would be maintained. After completion of the current step, the *CPU Server* would feed back an end signal and the *Executive Server* would start with the next part. However, this solution is not practical. First, it wastes scarce resources because the socket stream has to be kept alive all the time. If a legacy application would be executed for several hours or even some weeks, the ports have to be kept open and the resource of the computer are thus wasted on both sides. Second, this solution puts too much workload on

the *Executive Server*. The duty of the *Executive Server* is to manage the requests from all VADOR users, and not merely to focus on the details of each task.

Therefore, the suggested solution is to send the complete information of a task to the *CPU Server* in one single transaction, and then to cut the connection between the two *Servers*. The *CPU Server* analyses the information in details, controls all the procedures related to sub-tasks, and send back the final result to the *Executive Server* after the whole task has finished. This solution avoids the two main disadvantages of the previous one, but it invokes another issue: what is the best format for the message as it is being transfered? There are two formats available. The first is, once after the connection between the two *Server* has been established, the *Executive Server* send the information of every sub-task. Each time the *CPU Server* receives an information, it sends a signal back to inform the *Executive Server* to send the next one. At the end of the transfer, the *CPU Server* receives an "over" signal, and then it begins to summarize all the messages and execute the task. This choice doesn't require too much protocol between the two *Servers*. However it consumes tedious networking time to launch several transactions in order to transfer a single message.

To making the program more efficient and scalable, a prefered method is to put all the sub-tasks message into a unique object. Only one transaction is then enough. This method, however, needs the processes at each end to marshall and un-marshall the message objects. The message object has to be transfered in a single transaction as a whole, and it must be capable of describing any kind and number of requests, both simple and complex, internal and external, long and short, or even the combination of them all. On the *Executive Server* side, all the requests are put into a special format which is suitable for network transaction, the message is then sent to the *CPU Server* through a socket stream. On the *CPU Server* side,

a particular program will read and interpret the code with a special protocol, and then translate it into a series of sub-tasks that can be processed by the *CPU Server*.

3.2.2 Prepare the message object

The construction of a task package is done at the *Executive Server* side. The package should be scalable enough, so that even if there are significant changes in the content of the task information, it can still describe the task exactly and efficiently. The *CPU Server*, meanwhile, requires this package to be easily analysable by the interpretor program. The main objective of VADOR is to put all the necessary messages into a java "Vector" object through a predefined structure and protocol.

Figure 3.1 shows the architecture of a complete command. The first part identifies the style of this task: synchronous or asynchronous. The *CPU Server* decides how to return the result of execution from the style. The second part is the ID number of this task. In the *VADOR* project, this number must be kept unique while a task is being processed. It becomes significant only when the task is asynchronous. When the whole asynchronous task finishes, the ID number is the key used by the *Executive Server* to identify the returning result. The third part, and the most important, describes the detailed content of the whole task. There are several java vectors in this part, every vector encapsulates the working procedure of a sub-task. The order of the vectors is important, and must match the order that the *Executive Server* wants to follow.

There are two types of sub-tasks: "EXTERNAL" and "INTERNAL". Respectively, each type implements a different architecture of the sub-task vector regarding time consumption feature. The first element of the vector contains the string of the task type and attribute. Based on the first element, the interpretor program

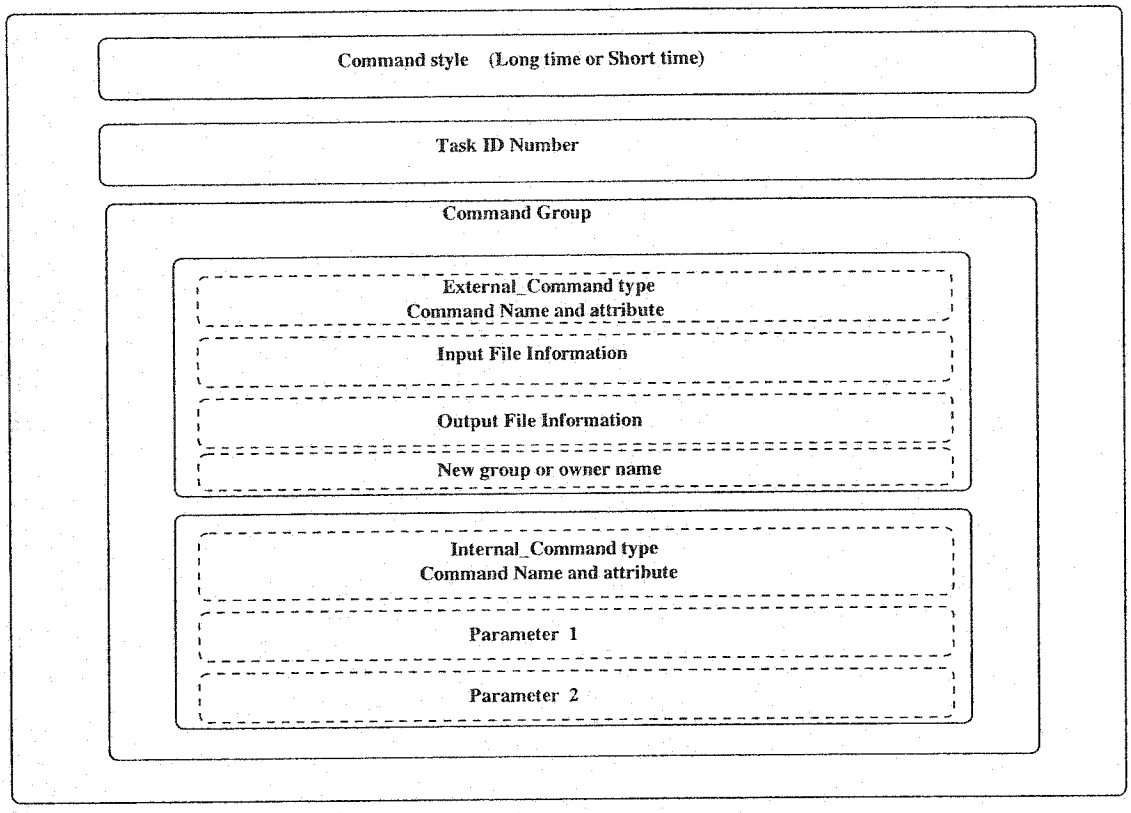


Figure 3.1 The structure of a command

decides how to read the rest. In the case of an external sub-task (executing a legacy application), the first element specifies whether this application should be sent to a remote machine that uses a queue to run its tasks. If so, the address of the machine is also included. The remaining elements are the detailed information about all the parameters. Each element is in vector format, the same way as the whole command object. If a parameter is a data file, its first element indicates the attribute of the file, such as "INPUT-FILE", "OUTPUT-FILE", "OPTIONAL-FILE", etc. Other attributess of the parameter specify the host and absolute path of this data file. Sometimes, arguments of a legacy application may be in string format, under this circumstance, the first element only contains the flag "OTHER-STRING" so that

the interpreter program can treat the rest element of the vector as a string.

If a sub-task is internal, the structure becomes relatively simple. The first element of the vector contains the name of the internal command. If it is a short-time command, the parameter has only one element, which stores all necessary informations. In the long-time case, such as moving or copying a file, the structure of parameters resembles a little bit that of external applications. The data files, however, do not have additional features such as "OPTIONAL-FILE".

The architecture of the whole command object is very scalable. It leaves room for additional parameters and command attributes for future request types. It also considers the possibility of expressing new types of sub-tasks. Because both simple and complex commands are described by the same structure, programs on the *Client* and *Server* sides can be modified as less as possible while some new kinds of commands are added into the request. However, this globally suitable solution has one blemish, that is, its structure is a little complex, which makes the programs needed to build and analyse its content also more complex.

3.2.3 Client side software architecture

The process of wrapping each component into a command object is performed by two specialized classes: "WrapperProxy" and "WrapperCommandBuilder". These classes, provided by the *CPU Server* package, are physically located and run on the *Executive Server* side. The "WrapperProxy" class is used to coordinate the communication between the two *Servers*. It exposes the necessary functions to express any current requests, therefore, the *Executive Server* calls upon these functions without having to deal directly with the details of building a command object. According to what the *Executive Server* intends to achieve, the "WrapperProxy"

class will invoke the proper "WrapperCommandBuilder" class to realize the final vector and send it to a proper *CPU Server*.

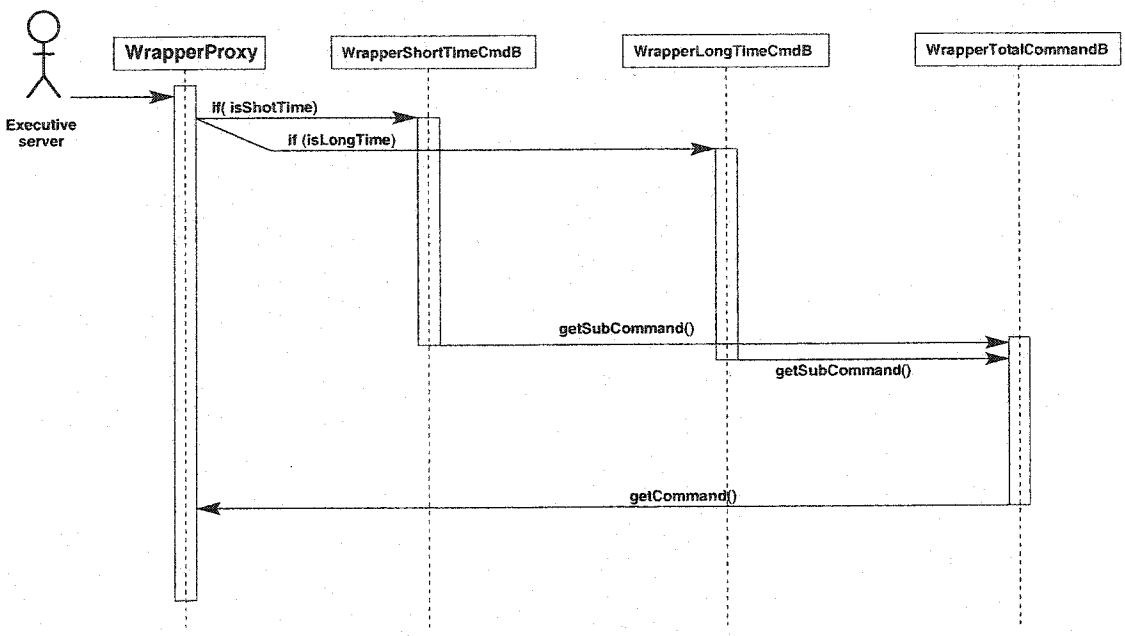


Figure 3.2 The sequential diagram of building a command

"WrapperLongTimeCmdB", a sub-class of "WrapperCommandBuilder", is in charge of organizing the descrption of all long-time sub-tasks because they share some similiar features such as setting the properties of input or output files. "WrapperShortTimeCmdB" aims to arrange all the information of short-time comsuming sub-tasks. When every message of sub-tasks is ready, they will be sent to "WrapperTotalCommandB", another sub-class of "WrapperCommandBuilder", to set the global attributes, like the ID number and style of the whole task. After, the total information vector is finished, "WrapperProxy" delivers it to the *CPU Server* side through the socket connection. The next step is to translate it and run the task.

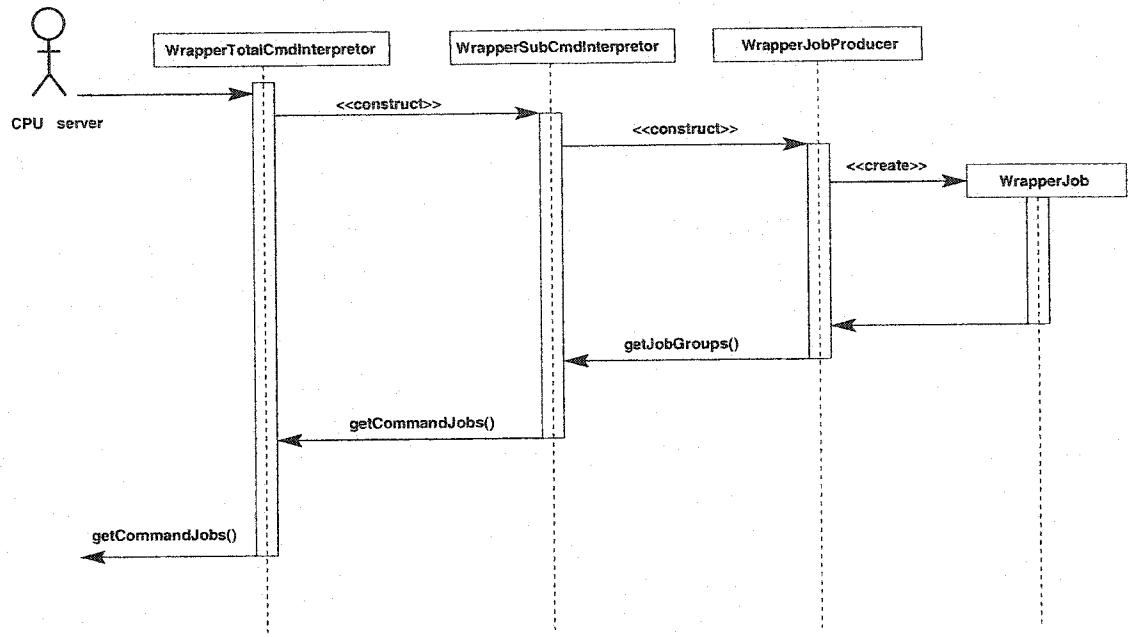


Figure 3.3 The sequential diagram of analysing a command

3.2.4 Server side software architecture

On the *CPU Server* side, the *CPU Server* keeps waiting for the outside requests which are in vector format. As a request arrives, the *CPU Server* (Wrapper program) begins to read and analyse the content of the vector. Since the vector can not be executed directly, the Wrapper program must convert the description into an executable task format. This work is conducted by the specialized classes "WrapperCmdInterpretor" and "WrapperJobProducer".(Figure 3.3)

The duty of the "WrapperCmdInterpretor" class is truely to perform the unmarshalling, for example, to peel off the layer of the vector and study its content. It has two sub-classes, "WrapperTotalCmdInterpretor" and "WrapperSubCmdInterpretor". The former gets the common information of the whole task, and invokes an object of the latter class to analyse the detailed information of each sub-task. The

goal of this process is to generate several executable jobs that can represent the total task. This process needs help from the "WrapperJobProducer" class. There are four children to this class: "WrapperExternalLongJobP", "WrapperExternalShortJobP", "WrapperInternalLongJobP", and "WrapperInternalShortJobP". Each of them produces jobs for a certain type of sub-task. After reading the content of the respective sub-task, the "WrapperSubCmdInterpretor" deploys the proper "WrapperJobProducer" to create the jobs.

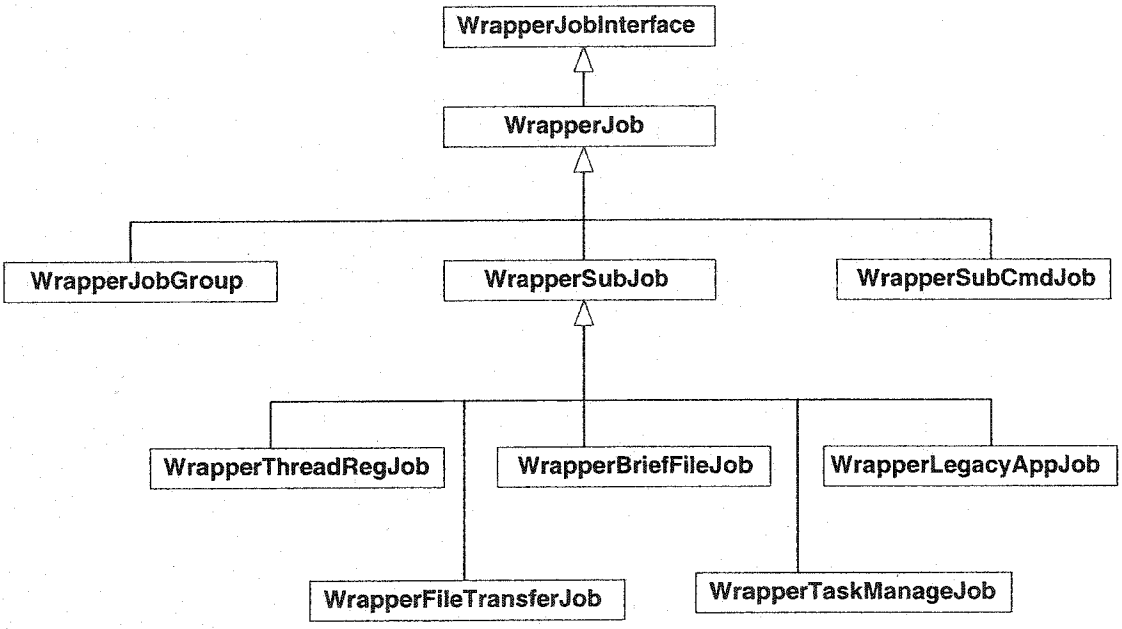


Figure 3.4 The WrapperJob class diagram

In the VADOR project, a job is an action that can accomplish a real work such as the transfer of a data file, execute a command string, remove a file, etc. Each job is an object of the class "WrapperJob" which can be understood and run by program of the *CPU Server*. The class "WrapperJob" has several sub-classes (Figure 3.4):

The main challenge of executing a task resides in the control of the execution flow. As presented above, a task may contain many sub-tasks. Inside each sub-task, there

are several execution stages, and every stage must be executed through at least one "WrapperJob" object. Jobs that can accomplish their purpose independently should be run simultaneously by means of threads in order to economize CPU time. On the other hand, the running results of some jobs may be the preconditions to other groups of jobs, which must then keep waiting until all former jobs have finished. Therefore, the total execution process of a task involes a mix of parallel and sequential jobs. The program on the *CPU Server* side must manipulate the time schedule accurately so that every job can be executed at the proper time.

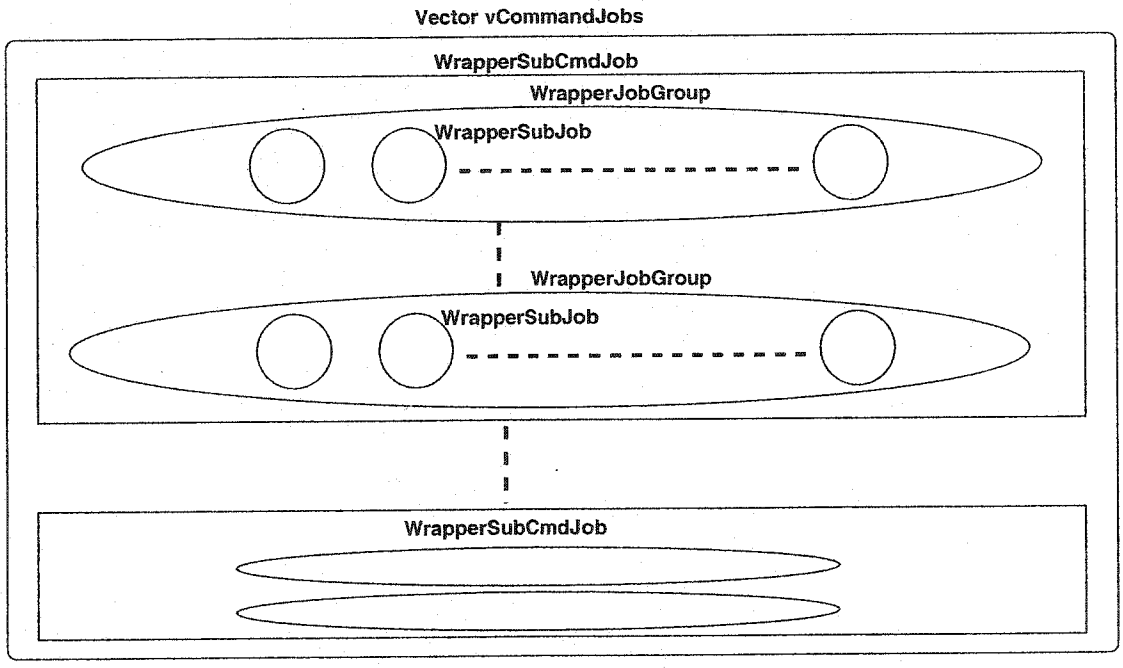


Figure 3.5 The structure of WrapperJob

The principle of time management can be stated as follows: no matter how many sub-tasks there are and how complex each task is, they can be translated into the same structure which contains only three kinds of elements implemented as three distinct classes:"WrapperSubCmdJob", "WrapperJobGroup", and "WrapperSub-Job". The "WrapperSubCmdJob" class controls the flow of the execution of a

single sub-task, the "WrapperJobGroup" class takes care of the jobs of each stage of a sub-task. Combining the effort of these classes with those of the "WrapperCmdInterpretor" and "WrapperJobProducer" classes, results in the responsibility structure illustrated in Figure 3.5.

The job vector comprises objects belonging to the "WrapperSubCmdJob" class which implements sequential execution order of its elements. Each "WrapperSubCmdJob" object contains a "WrapperJobGroup" object whose execution order is also sequential. In the "WrapperJobGroup", there are several "WrapperSubJob" objects whose execution order is parallel. This architecture accurately expresses the time schedule of the total jobs, the only thing left is to run all the jobs according to the schedule.

3.2.5 Job processing

The final responsibility of the *CPU Server* is to perform all the real work associated with an incoming task. This duty is carried out by the "WrapperTaskExecuter" class. Two sub-classes of this class have been defined, the "WrapperShortTaskExecuter" and "WrapperLongTaskExecuter" classes, which are repectively used to treat synchronous and asynchronous tasks. From the task style, the *CPU Server* can choose suitable object of the "WrapperTaskExecuter" class to construct in order to treat a given task.

The purpose of this step is to coordinate every job and task. In the synchronous tasks case, this responsibility is relatively simple because the executing order of all the jobs is only sequential. In the asynchronous case, the procedure becomes more complex. First, because the sequential jobs are mixed with the parallel jobs, the *CPU Server* must determine the point where one group of jobs finishes, so

that it can invoke the next group. Second, on each machine, there may be many tasks running at once, every task including many jobs (sequential or parallel). The *CPU Server* must identify the owner of the jobs when they signal their end. Third, sometimes, the parallel jobs are implemented by threads: for instance, transfers of input or output data files. The threads may be launched on local or remote hosts. In the case of remote threads, the jobs still need to report their result. The *CPU Server* should use a different method to treat the information from them.

A solution is to set up a series of threads, each thread controling one task. The thread stores all the necessary messages associated with its task, and keeps alive until the task is completed. The duty of the *CPU Server* then becomes management of these threads. The main idea behind this solution is to utilize a timer function in the thread. The task thread checks the status of the current jobs periodically, and if it finds that all the current jobs have finished, it launches the next group of jobs to execute. However, this solution is not efficient. In fact, the execution time of each task may be very different, ranging from seconds to weeks, and we know that a thread occupies more memory than a common object, therefore, if every task need very long time, it is not a good idea to keep the threads alive during all the task execution time.

A better solution to implement this aspect of the *CPU Server* is to develop a "call-back" system. This system maintains three data structures, which separately manage the three parts of the status of the tasks: threads management (ThreadsControler), execution status (TasksControler) and result of jobs (ResultsControler).

The responsibility of "ThreadsControler" is to manage communication mechanisms between all active threads of the tasks on the Server. (See the structure in Figure 3.6).

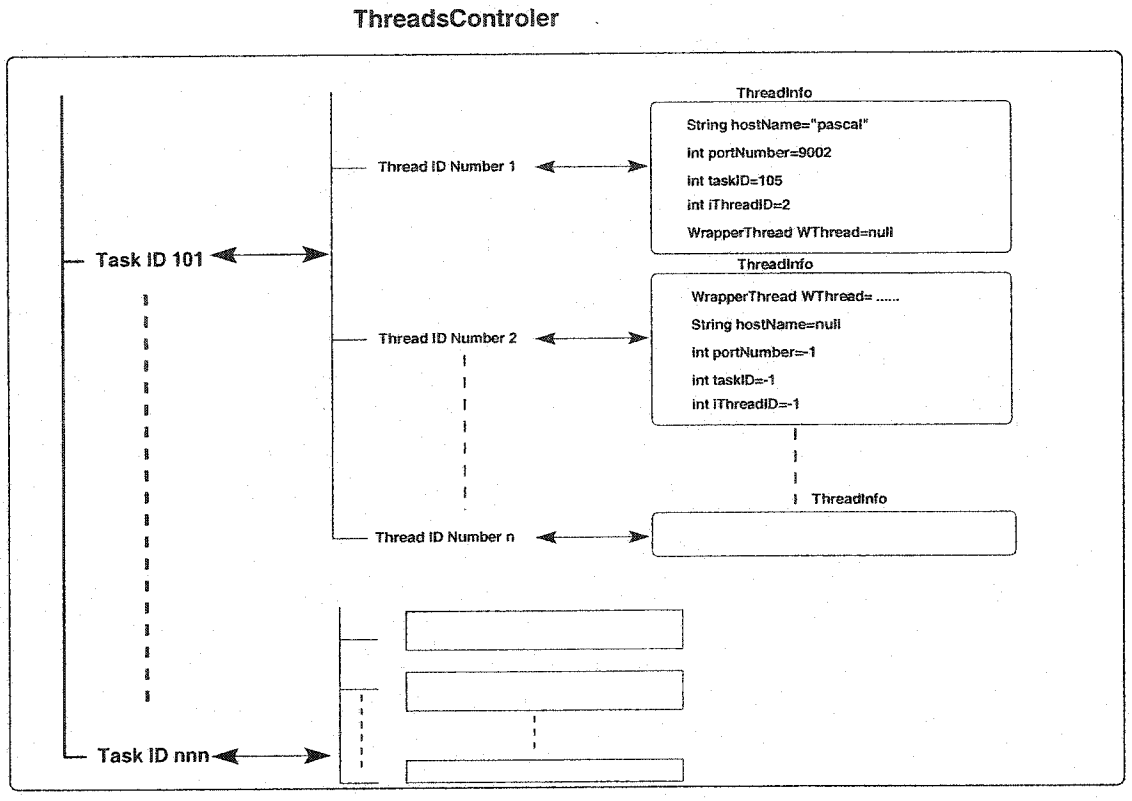


Figure 3.6 The structure of ThreadsControler

In the VADOR project, the threads are able to inform the *CPU Server* about their status. When a thread finishes its job, it should report its completion. If it meets a problem, it must send a message back to the *CPU Server* to describe what happened. Then, the *CPU Server* may decide whether it will launch the next step of the task or teminate the execution and report the error message to the *Executive Server*. Every thread must thus carry information on how to communicate with the *CPU Server*. On the other hand, the *CPU Server* also needs to contact the threads when necessary. For example, if a problem happens in a task, and the *CPU Server* needs to shut the task down, it becomes unnecessary to keep the remaining threads of the task running; the *CPU Server* should then be able to find those threads and

instruct them to stop early. The communication between the *CPU Server* and its threads should therefore be bidirectional.

Two distinct types of threads can be launched by the "ThreadsControler" class. The first type, which is the most common, is directly launched locally on the same machine; but some threads must be launched on remote servers, for instance to perform uploading. For each of these type of threads, the information stored in the "ThreadsControler" are different. In the remote case, every thread launches a small *Server* called "ThreadConnector", that keeps ready to accept any message from the outside. The "ThreadsControler" then stores the connector data information that includes its host address, port number and thread ID. When the *CPU Server* needs to connect with that thread, it will load those informations and build a socket connection to send the command (look at the thread 1 in the Figure 3.6). In the local case, this procedure becomes simpler, the "ThreadsControler" only keeps the handle of the local thread, the *CPU Server* is then able to directly call functions on the thread to control it (look at the thread 2 in the Figure 3.6).

The structure of the "TasksControler" is simpler than the "ThreadsControler". Its main duty is to provide a convenient way to manipulate executing tasks. (See the structure in Figure 3.7)

When a new task comes in, if it is asynchronous, the *CPU Server* registers it into the "TasksControler" class. In Figure 3.7, each leaf of the "tree" contains the handle of a "TaskExecuter" which is managing the execution of a specified task. In the "call-back" system, the *CPU Server* needs a service to control the process associated with each task. The most important responsibility of the "TasksControler" class is to instruct each task to run the appropriate job at the appropriate time. Also, the "TasksControler" should provide a way of influencing the work status of a task, for example, to abort, pause or resume the running process.

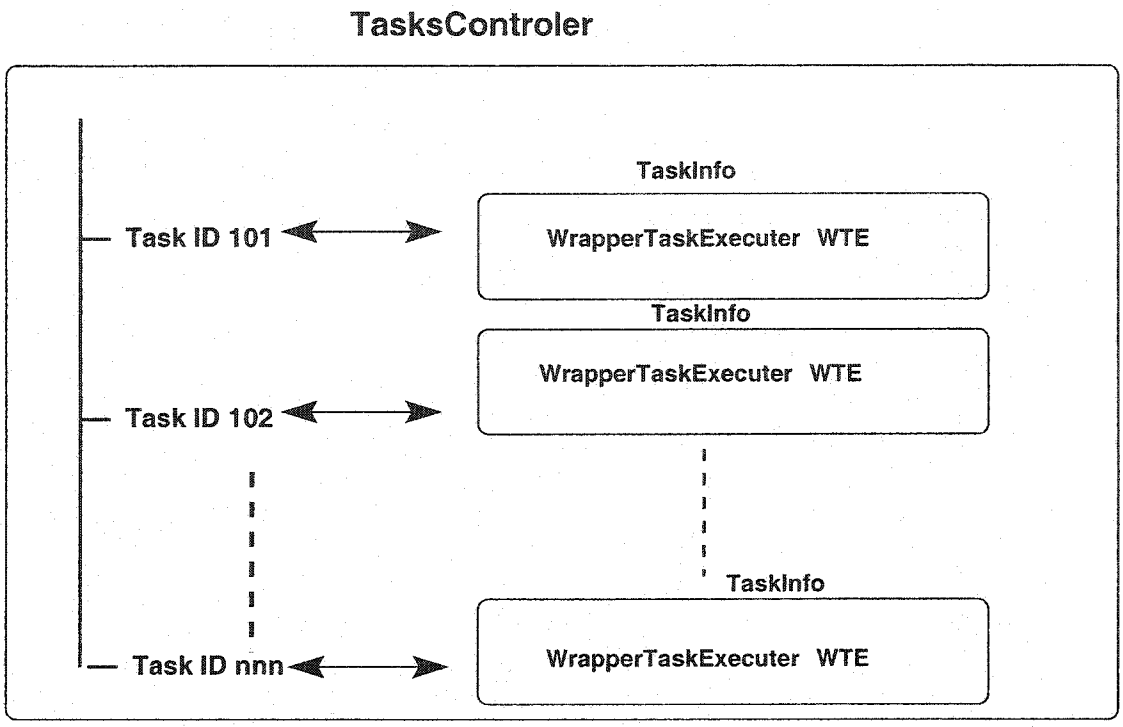


Figure 3.7 The structure of TasksControler

The "ResultsControler" records all the results of a given task. When a job finishes, there must be a result that gets created. If no problem happened, the result contains a successful flag, otherwise, it contains the detailed error message which will be displayed to the administrator or the user so that they can know what went wrong. According to these results, the *CPU Server* decides if the working process should go to the next step, or be aborted. Therefore, the "ResultsControler" is the most important part of the "call-back" process management. For this reason, every job, either local or remote, short or long, must register its result. Moreover, since the result of a group of jobs may be used by the next group, the "ResultsControler" provides a good place for those jobs which want to look up the previous results. (The structure of the "ResultsControler" is shown in Figure 3.8).

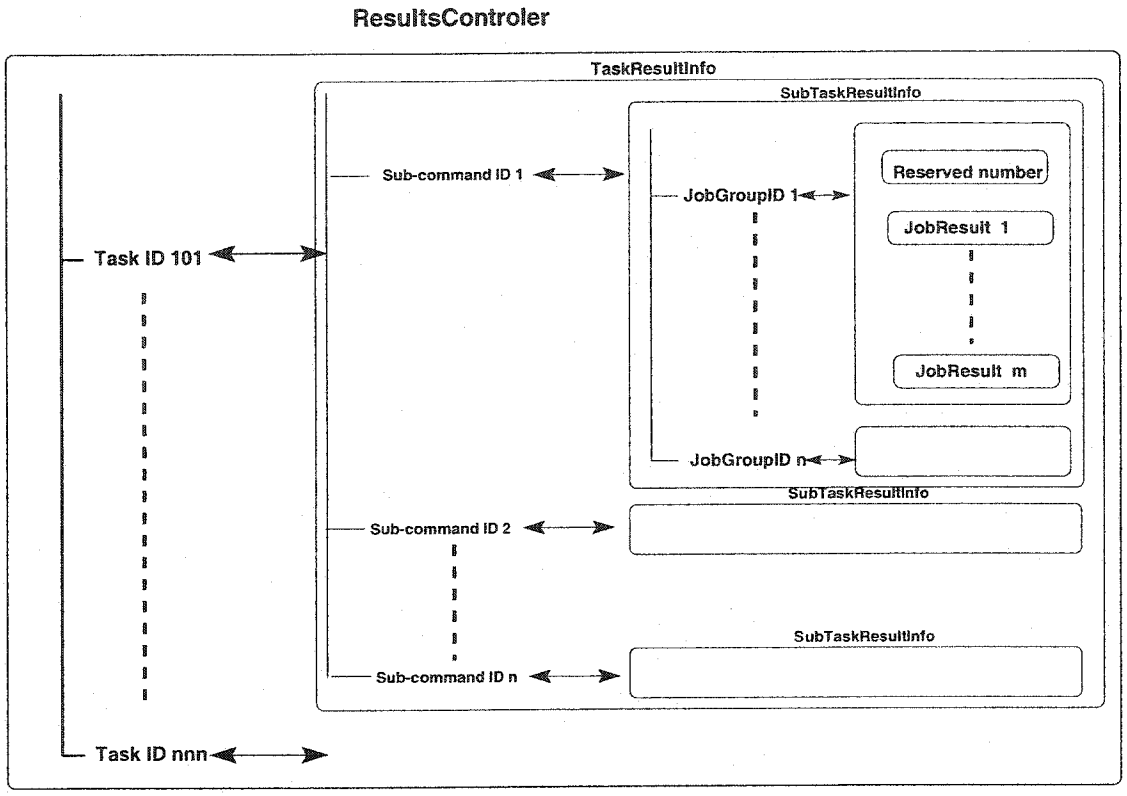


Figure 3.8 The structure of ResultsControler

The content of the "ResultsControler" is more complex than the former two classes, it includes four layers, each layer corresponding to a level in the "WrapperJob" structure. In order to allow completed jobs to register their results correctly, the *CPU Server* assigns ID numbers to all the sub-tasks and job groups. Every job should carry enough information for the registration of its result. When a result comes in, the *CPU Server* first checks the content of the result. If no error message is found, the result will be handed to the "ResultsControler". After the result is registered to its specified destination, the "ResultsControler" tests if all the current parallel jobs are finished, and returns a signal to the *CPU Server* so that it can proceed with task execution.

The next issue is how to determine wether a group of jobs has really finished. The number and duration of the jobs which are launched by a sub-task are variable. Thus, the "ResultsControler" should have a mechanism to detect the working status of each group of jobs. In each job group, there is a number that is reserved for counting the future results. Before the *CPU Server* invokes a group of group of threads for running the jobs, it sets the total number of jobs. When a job registers its result, this number is decremented. The only thing that the "ResultsControler" should do is to check this number, if it becomes zero, the gate to the next step opens.

With the help of the previous three classes, the "TaskExecuter" works step by step according to the time schedule which is carried by the "WrapperJob". When there is not further step to go, the whole task is finished, the *CPU Server* will report the final result to the *Executive Server* and log this task out of the memory.

3.2.6 A special instance of legacy application – optimizer

In Bombardier, the aeronautical design includes several stages. For every stage, some simulation codes have been developped to help the engineers preview the design. The engineers need optimization methodologies to integrate the calculations of all those legacy codes, so that they can obtain the best decision under various constraints. For example, to design a wing profiles for transonic flight, an optimizer should be used in order to minimize the value of a function F(x), such as drag, under a certain set of constraints G(x) such as minimum lift, minimum thickness, maximum span, etc.(see the Figure 3.9)

In this case, getting the value at x^* is the goal of the optimizer. In this case, the general approach to solution is to iteratively execute a set of legacy design

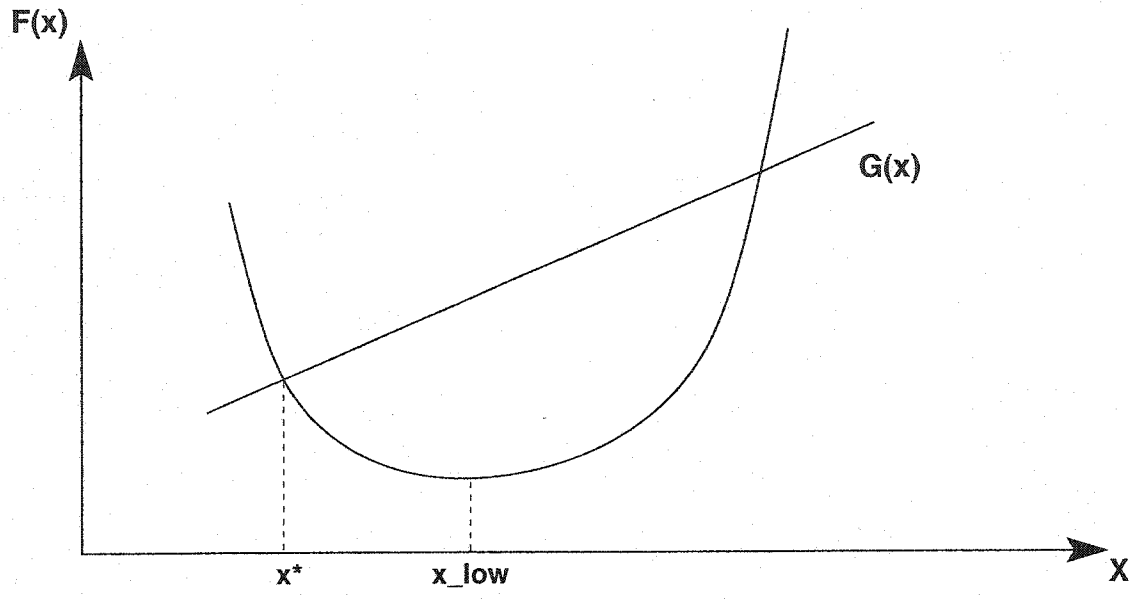


Figure 3.9 Function for wing shape design

applications. At each execution, the optimizer provides different input data. The application iterates several times over the calculations to approach the best results.

An optimizer is a particular type of legacy application because it can invoke some other legacy applications while it is running. Only once the external applications have finished, can the optimizer go on to perform the next calculation. The challenge of realizing this procedure in the VADOR project involes maintaining the relationship among the optimizer and its legacy applications. In the VADOR system, every application should be handled by VADOR so that their processes can be coordinated. Thus, when the optimizer needs to run an external application, it should ask VADOR to launch it rather than launching it itself. However, sometimes the optimizer is itself written in a legacy computer language such as Fortran, or even in script language. These codes are thus extremely weak in network communication, and it is often impossible to modify then so that they can send the necessary

information to VADOR in order to execute the target applications. Even if language with strong networking functionality were used, engineers from Bombardier can not be expected to write complex code for every optimizer to communicate with VADOR.

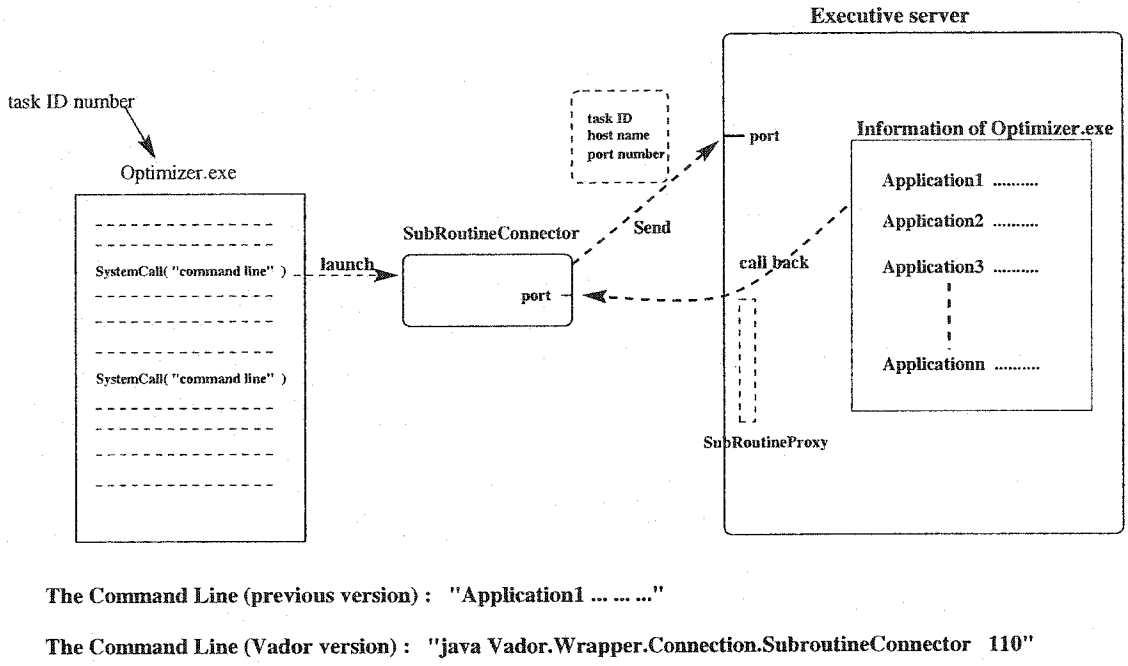


Figure 3.10 The optimizer's working procedure

To solve this problem, the *CPU Server* provides a class called "SubRoutineConnector". Its main purpose is to connect back with the *Executive Server* to send some information about the current application, such as the ID number, and working step of this task, and then to keep waiting for the end of the application. When the optimizer needs to start an external calculation process, it sends its request to the *Executive Server* through the "SubRoutineConnector".(see the Figure 3.10).

Before the optimizer was launched, the user specified its whole process by means of the VADOR GUI. VADOR keeps these message in its database and assigns a

universally unique ID to the optimizer. If an execution request of an optimizer is submitted, the *Executive Server* identifies the ID number, and checks the associated data record. When it finds the description of the next working step, it asks a proper *CPU Server* to run the task. As soon as this task finishes, the *Executive Server* informs the "SubRoutineConnector" immediately. Then, the "SubRoutineConnector" terminates its execution life, the optimizer can thus detect the end of the external calculation and try to read the result. In this scheme, the only thing that the optimizer should change, compared with its normal mode of operation, is to add a parameter for its ID number, and pass it to the "SubRoutineConnector".

This solution involves the cooperation of all the parts in the VADOR system. It liberates the engineers from making complex networking code and memorizing tricky protocols. It also sufficiently utilizes the service of the VADOR programs. During the execution of the optimizer, VADOR still handles the information for every application, thus it can distribute computer resource as suitably as possible.

3.3 The Mobile Agents based solution for the CPU server

This solution utilizes a large number of advantages brought by mobile agent technology to implement a working prototype of the *CPU Server*. It supposes that every machine is equipped with a mobile agent platform. The actor who is in charge of a task is a group of agents rather than a set of stationary codes which are already deployed on each computer. Because most of the communication work load is taken incharge by the platform, an agent is more easier in making contact with other agents than before. This conveniently allows the code of the agent to focus on executing a task, and releases it from protocols and information transfer duties. Thus, the programming of this solution is more concise and flexible than

the previous one.

3.3.1 Mobile Agents platform and protocols

To guarantee a good working environment for mobile agents, an open-standard and stable platform is absolutely necessary. The main duty of the platform is to maintain the status of all the local agents and deliver the messages among the agents and platforms. As discussed in the second chapter of this document, there are several standards which can be selected for the communication. Among them, MASIF and FIPA are the most widely used in both commercial and academic areas. This section briefly describes the principles of the agent platforms and communication protocols that are pertinent to the VADOR project.

1. The architecture and organization of common agent platforms

An Agent platform is a system that is based on an operating system (such as Windows and Unix) of the local machine. It can create, interpret, execute, transfer and terminate agents. Dependsing on the programming languages, serialization methods and manmufacturers, agent platforms are divided into many types. The profile of an agent must specify the type of mother platform, so that the new host be able to identify the required fuctionality needed to manipulate this agent.

Figure 3.11 shows the organization of MA platforms. To run an agent, every MA platform must provide an execution environment called *Place*. The duty of *Place* is to offer services such as access control. One MA platform may have several *Places*. The agents actually migrate among all the *Places* rather than platforms. During their migrations, the departure *Places* and the destination *Places* can be located on either the same or different platforms. If the platforms are different, they must support the same agent profile.

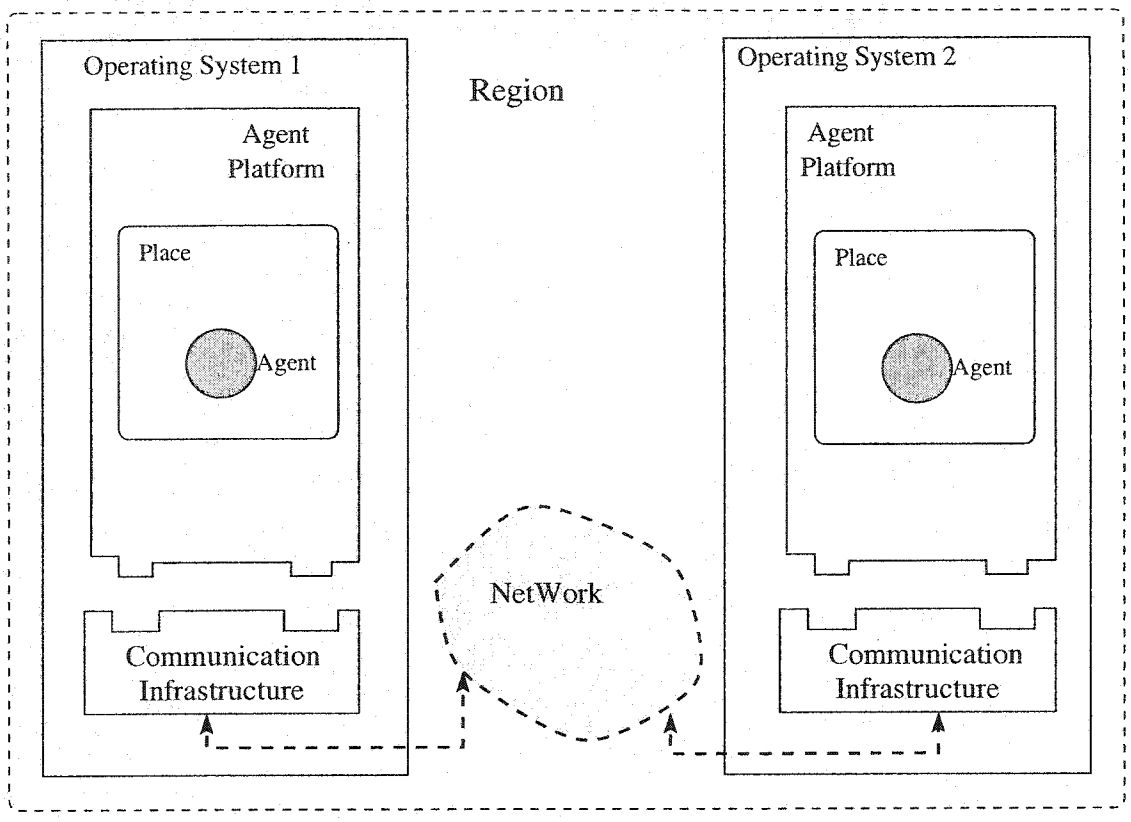


Figure 3.11 The architecture of Agent Platform

An important issue of MA platform is authentication. Since many agent platforms are running without human supervision, they need a mutual authentication to satisfy their security policies. The MA platforms which have the same authority collectively share a certain level of trust in exchanging agents. They can be organized in a *Region* which allows a group of MA platforms to represent the same person or organization. Therefore, *Region* becomes another element in an agent's location, anybody who wants to contact an agent must know the following three localization informations: the address of the *Region*, the name of the *Place* and the identification of the agent.

A *Region* may contain many agent systems (platforms). Some of them are exposed

to the outside whereas other systems are only accessible from inside the region. The externally accessible systems are defined as region access points. Any outside agent system or *Client* which needs to contact an internal agent system must go through these points, just like passing a firewall. In a *Region*, the interconnection between two MA platforms is implemented through a *Communication Infrastructure* (CI). A CI provides the services of communications transport (the method for transfering messages, for example, Remote Procedure Call or Socket), naming (construct unique identification for agents or platform) and security (identify the mutual authentications with other MA platforms).

2. MASIF compliant MA system:

In chapter 2, MASIF was presented as a standard which aims to promote interoperability and diversity of MA systems. It doesn't want to standardize local agent operations such as agent interpretation, serialization or deserialization. Rather, it focuses on the agent system level more than agent level. The following four aspects are defined by the MASIF standard: first, agent management, which is in charge of managing all different types of agents via standard operations. Second, agent transfer, which lets agent applications freely move among all kinds of agent systems and result in a common infrastructure. Third, agent and agent system names, which provides a standard syntax and semantics of the names, so that the agents and agent systems can identify each other. Fourth, the agent system type and location syntax, which can help the agent systems to locate each other.

The MASIF standard strongly recommends CORBA services as the base of MA system CI. There are four services that can be utilized by MA system. The first one is the CORBA Naming Service which binds a globally unique name to a CORBA object. An agent which acts as a CORBA object may publish itself using the Name Service. Other applications who want to contact this agent can search and

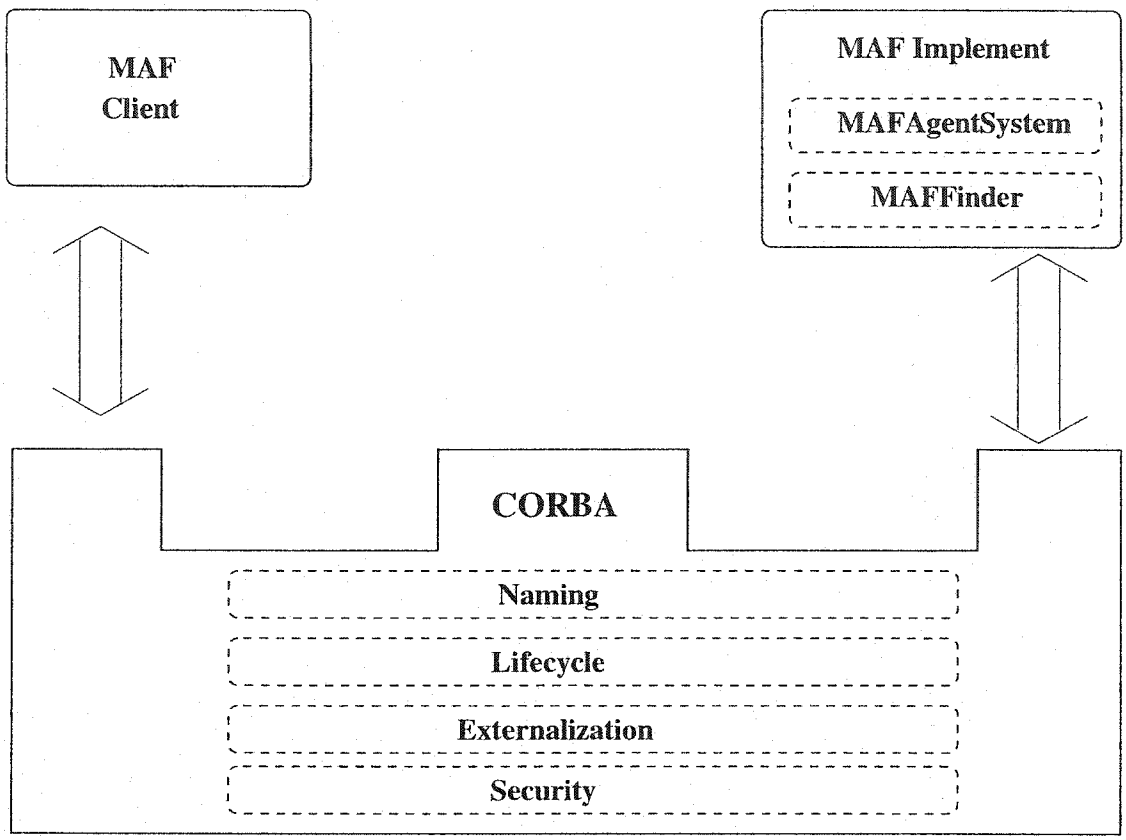


Figure 3.12 The architecture of MASIF compliant MA system

obtain its reference by providing only a name. The second service is the Life Cyle Service, which defines services and conventions for creating, deleting, copying and moving CORBA objects. Agent systems may use this service to control the lives of local agents or migrate them. When a CORBA based agent is moving, its state should also be transfered, this needs the help of the third service: Externalization Service. This service provides a standard mechanism for packing an object's state onto a data stream, and for restoring the state from a data stream. The last service provided by CORBA is the Security Service which meets almost all security needs of MA systems. Its main part is to define authentication among the *Client* applications, agents and agent systems.

Although MASIF compliant MA systems can implement many functionalities through CORBA services, they still need to define their own interfaces to satisfy some special requirement. Because CORBA services are designed for static objects, they are not suitable to handle all aspects the case of mobile agents. Therefore, the MASIF standard defines two interfaces, MAFAgentSystem and MAFFinder, to complement the disadvantages of the CORBA services with respect to agent management and agent naming.

3. FIPA compliant MA system:

Like MASIF, the core mission of FIPA is to facilitate the interworking of agents and agent systems across multiple vendor's platforms. FIPA focuses on standardizing the communication messages that are transfered among agents and agent systems, in order to provide a standard mechanism to express the intended communication semantic accurately.

Figure 3.13 shows the abstract architecture of a FIPA compliant MA system. In each FIPA agency, there are two special agents, DF and AMS, which are ready to offer agent management services. DF provides a "yellow pages" service. Agents or external applications may query the DF to gather information about other agents. AMS provides a "white pages" service, which include creation, deletion, registration of agents on the local platform and overseeing the migration of agents to and from other platforms. Moreover, AMS maintains an index of every agent which is currently registered on the platform. Each agent is identified by its name which is composed of the agent's place, platform address and a unique identifier within that platform.

In FIPA compliant agent systems, agents communicate among themselves by sending messages through the agent communication channel (ACC), which is a mech-

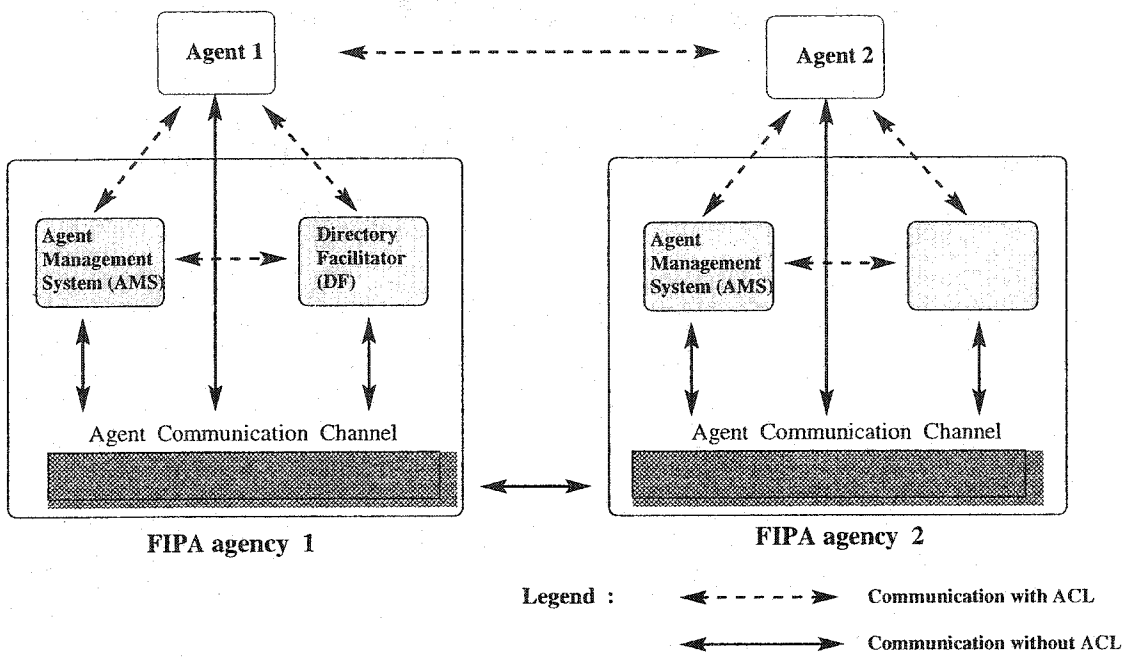


Figure 3.13 The architecture of FIPA compliant MA system

anism for delivering these messages. The FIPA specification has standardized the meaasge structure. A message should specify the name of the sender and receiver agents. Its content is written in FIPA ACL. The message must be encapsulated into a payload and then an envelope so that it can carry enough information for network transport and security. The purpose of the payload is to prevent the message content from being stolen. Agent systems may encrypt the payload before sending a message and restore the content when it arrives.

4. Summary

MASIF and FIPA standards have similiar goals, but they do not conflict with each other. The structures of their compliant platforms have the same features.The only difference is that MASIF emphasizes low level communication management, whereas FIPA takes care of the top level services. The MA solution considered for

the VADOR project aims to utilize a MA platform which is the compliant both to MASIF and FIPA, so that it can take advantages of standardization efforts introduced by both specifications.

3.3.2 Agent-based remote application execution overview

Execution aspects of a *MA-based CPU Server* solution differs totally from the *C/S* solution. In the case of *MA*, only one copy of the *CPU Server* code is necessary. It is deploied on the machine that hosts the *Executive Server*. Depending on the content of the tasks that must be executed, the *CPU Server* prepares a series of prototypes for the various working agents needed to perform task executions. When a task comes, the *CPU Server* propagates several agents based on the prototypes, and assigns missions to each of them. These agents then go to a proper machine and begin their job under a particular order. Finally, when this global mission is finished, the agents report the results (if the execution is successful) or error message (if a problem happened) back to the *Executive Server*, and then remove themselves from that machine.

The execution process comprises three distinct phrases, namely 1) job preparation, 2) local agent launch and 3) agent migration and task execution.

1. Prepare the jobs

Since the *CPU Server* and the *Executive Server* are located on the same machine, it is not necessary to transfer the task command over the network. The procedure of serializing or deserializing command can thus be ignored. In the MA solution, a new mission then begins through the invocation, by the *Executive Server*, of the "executeCommand" method of the "WrapperMAProxy" class. It emploies the "WrapperTaskDeployer" class to collect all the necessary jobs according to the

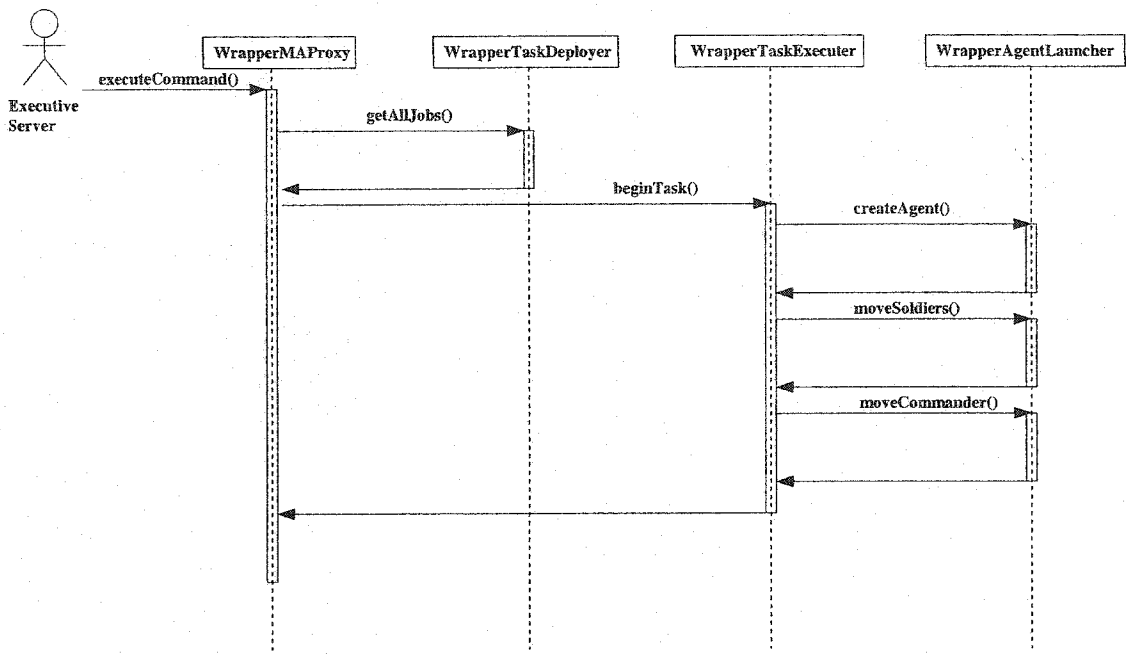


Figure 3.14 Prepare jobs for Mobile Agent model

content of task command. As discussed above, a task is composed of several groups of jobs which have a certain executing order. To manage the execution process, first, the "WrapperTaskDeployer" class analyses the commands, and then divides them into a series of jobs. Next, these jobs are sorted/stored into several vectors, from which jobs are executed and performed simultaneously. All the jobs in a vector will be executed by the agents of the same type. Therefore, the command encapsulated into a data structure can easily be read by the classes in the following step.

2. Launch agents on the local host:

Once the tasks have been stored in vectors, the *CPU Server* begins to launch the necessary agents on the basis of the job class. It has to face challenges similar to those encountered in which is the *C/S* model: how should the *CPU Server*

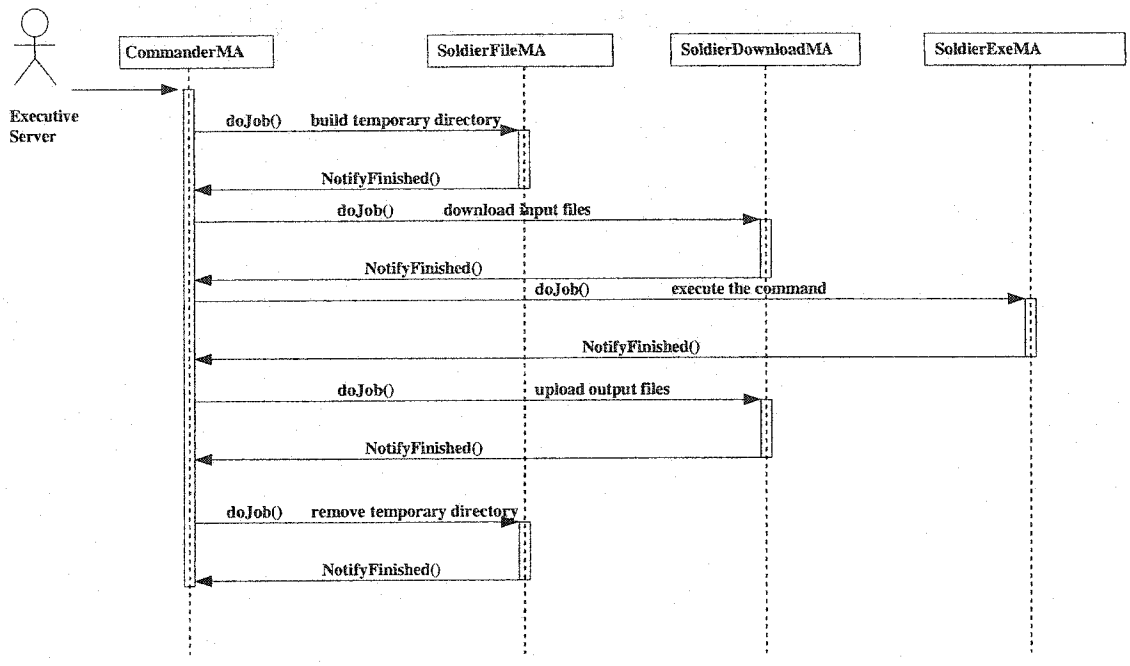


Figure 3.15 Process jobs in Mobile Agent model

manage the execution of a group of parallel jobs? In prototype solution of an agent-based *CPU Server*, it has been decided that every job should be done by an autonomous agent. The *CPU Server* has two types of agents, "commander" and "soldier" agents. A "soldier" is an agent which is able to accomplish a limited number of specific tasks. For example, the "SoldierDownloadMA" manages file transactions between two machines that do not share disk access through a single NFS domain; the "SoldierExeMA" focuses on executing a legacy application, and the "SoldierFileMA" is in charge of all the manipulation of local files such as setting up a directory or removing a file. Different from the "soldiers", the "commander" agent doesn't perform any real job, its duty is just to coordinate the behaviors of all the "soldiers".

If a task contains only one sub-task, or the processing of the task does not involve

any time order control, the *CPU Server* will send only one "soldier" agent to accomplish the mission. If the task, usually an asynchronous task in that case, needs the help of several sub-tasks, the *CPU Server* then recruits a group of "soldiers" and a "commander". Each "soldier" deals with one sub-task whereas the "commander" carries overall time schedule of the execution process.

A second issue that must be discussed is how to manage the relationship between agents and jobs? One simple solution is that each agent takes care of a simple job. In other words, for every job, the *CPU Server* will create an agent instance from the agent prototypes to execute it. This solution is simple to implement, but it may consume significant amount of bandwidth when these agents are delivered to the target MA platform. For example, if a task needs to download or upload a total of 20 files, the *CPU Server* then must send 20 times the same kind of "soldier" agents which only carry the desired task for different file paths on the remote machine. The *CPU Server* may take a more efficient solution: before it launches the agents, the *CPU Server* summarizes the jobs into several types of tasks, and then deploies only one "soldier" agent for one type of job. On the target machine, because one "soldier" can not perform parallel jobs efficiently, the "commander" will order the "soldier" to copy itself to get the required number of agents. That is, the *CPU Server* only sends "soldier mothers" to the remote MA platform, and thereby decreases the burden on the network. The only inconvenience to this solution is that the "commander" needs more complex code to assign proper jobs to the agents after they are breeded.

3. Migrate to remote hosts and execute:

In this step, if the *CPU Server* needs to conduct an asychronous task, a group of agents should be sent to a remote place to work. If some result data files of this task need to be put on other machines which are located outside of the cur-

rent NFS domain, some "soldier" agents should go to pertinent host to perform "uploading". We know that in the *C/S* approach to remote execution, VADOR's uploading approach is the opposite of downloading, thus the *CPU Server* needs to send a "SoldierDownloadMA" agent to the target machine. There are two ways for sending this agent. In the first approach, every agent must migrate to the machine where the tasks should be executed, and then when it is necessary to upload files, the "commander" will send the "SoldierDownloadMA" agents to the places where they are ultimately needed. A second approach consists in sending the "SoldierDownloadMA" agents directly to the right machines when the "commander" agent is on its way to the task executing host. In the prototype version, the *CPU Server* implements the second approach because it is more efficient, although it is less convenient in terms of agent management aspect, which will be discussed in the next paragraph.

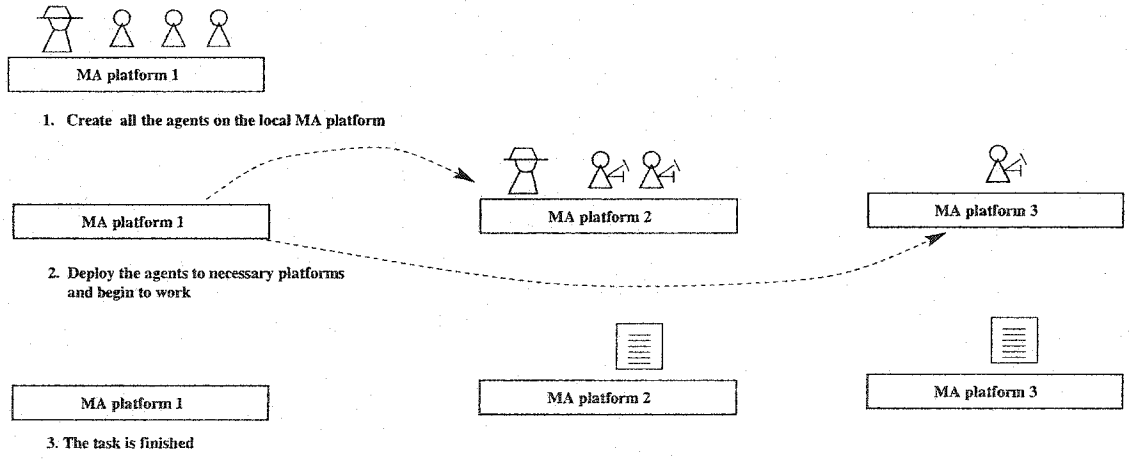


Figure 3.16 The working procedure of MA system

Obviously, on the remote machine, the "commander" agent plays a very important role in running the task because it is the only agent that has an overall knowledge of the complete set of execution steps. As soon as the "commander" reaches the

target host, it begins to look for its colleagues who will perform for the first step. The *CPU Server* must make sure that all the "soldier" agents have already arrived when the "commander" needs to contact them, so the "commander" is always the last agent to depart from the original machine. However, this solution can not completely guarantee that all necessary agents are ready, especially if there are some agents which are sent to other hosts to accomplish the uploading jobs. Therefore, each "soldier" agent is equipped with a "ping" function that can be used for detecting its status. When a "soldier" agent finishes its job, it will send a message to the "commander" agent. The latter will evaluate the execution result and decide if work can move on to the next step. The "commander" will also analyse the remaining jobs and decide wether it should remove "soldier" agents from the current MA platform. When all jobs are finished, the "commander" agent will informs the *CPU Server* and then kills itself. Thus, the task is completed on that host.

To guarantee that every "soldier" agent can work accurately on time, the communication between "soldier" and "commander" agents must be sufficiently reliable. The MASIF standard sets up a good "hard environment", an uniformed message exchanging way, to all the agent platforms. Any agents which work on MASIF compliant platform send and receive information through CI, we are not necessary to care which programming language the platforms and agents are using. Therefore, it is possible to build a globally suitable communication among all kinds of agent platforms in the world. The FIPA standard provides an excellent "soft environment", an globally comprehensible language, to all the agents. If the "soldier" and "commander" agents are located on different kinds of platforms, as long as the platforms conform MASIF and FIPA standards, the communication between them is still robust and comprehensible.

3.4 Summary and comments on the two approaches

In the VADOR project, both the *C/S* and *MA* solutions can accomplish the same tasks. The *C/S* model is a relatively mature technology. The programs are deploied on every machines, they act as either *Server* or *Client*. All messages related to task execution are transfered through socket streams. However, This solution must confront two main challenges: how to send the complete information about a task in a single network communication, and how to manage the execution order of sub-tasks within the task. In order to meet the first challenge, VADOR encapsulates the description of a task into a vector, and provides a mechanism to interpret the vector. This method needs a predefined structure and protocol between two *Servers*. As for the second challenge, the *CPU Server* prepares three data structures to manage the execution steps of all the tasks, threads and execution results. Each sub-task reports its status via a *call-back* mechanism. This solution works robustly and efficiently, but it can not avoid facing bottlenecks while transfering messages. Moreover, the *CPU Server* must be installed on every machine so that they can use the local resources available to execute the incoming tasks. This deployment might not be very convenient if developers often need to upgrade the codes. Up to now, most implementations of distributed applications based on the *C/S* model face similar problems, and in that respect, the prototype version implemented for the VADOR project is not different.

On the other hand, *MA* approach is a developing technology. In this model, the program may be located and deploied on a single computer. When the *Executive Server* needs to utilize the resources of a machine to perform a task, it only needs to send a request to the *CPU Server*, asking to send a set of mobile agents to execute the commands. The communication between the *CPU Server* and the agents is managed directly by the MA systems on both sides. Many efforts are being made

to standardize the message formats and communication mechanisms. The main advantage of this solution is to economize design, development and deployment time on the part of the developers. Because the agents can carry task information and directly interact with the application on the target host, the developer does not need to write code to solve the bottleneck problem. Besides, since the agents can remotely represent the *Executive Server* and supervise the execution process, the architecture required to store the intermediate results is also omitted. However, these conveniences are made possible only in the context of a stable and open-standard agent platform. So far, many platforms are still under development and none is very stable. In the future, when the MA platforms become commodities found on most every operating systems and as they grow more robust, people may totally enjoy the advantages that MA solution brings.

CHAPTER 4

APPLICATIONS AND RESULTS

For its production mode implementation of the remote execution servers, the VADOR project uses the *Client/Server (C/S)* as its distribution model because of the maturity and stablity of this solution. As soon as the user sets up a calculation procedure and submits his execution request through the Vador GUI, the executive server takes control of the process, translates the command into several different tasks, and then sends each task to the proper *CPU servers* for calculation. On each remote machine, there is a *CPU server* running. The main resposibilities of the server are launching legacy applications and performing file management. Depending on user requests, the Executive server may deploy tasks in different orders, for example, in parallel, sequentially, or in a combination of the two. This chapter presents the execution process of a real calculation, using an optimizer, to show how the *CPU server* can cooperate with the Executive server. Of course, the *Mobile Agent (MA)* version of the CPU server can accomplish the same tasks as the *C/S* version. The main purpose of this chapter is to compare these two approaches on a concrete application and illustrate their merits and disadvantages.

4.1 The Airfoil Shape Optimization:

The preliminary design of an airplane includes several parts, and one of them involves the optimization of wing profiles. The goal of this process is to modify the shape of the wings, and thereby improve their aerodynamic performance. In order to set up a wing profile optimization problem, engineers must specify a number

of design variables which depend directly on the geometric representations of the wing(Bézier, B-splines and NURBS). The sample process used in this chapter is based on a Non Uniform Rational B-Spline (NURBS) representation of the geometry, which allows for flexibility of representation and simple modification.

4.1.1 Optimization Principle

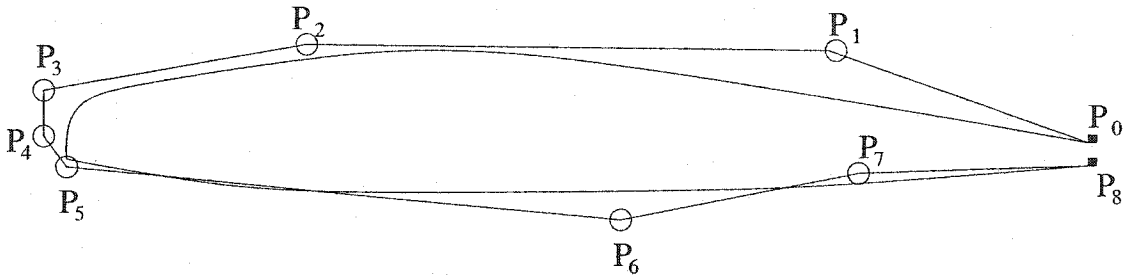


Figure 4.1 NURBS and Discrete points approach

Using B-spline curves, experts at Bombardier and CERCA have developed a complete methodology for the representation and optimization of wing profiles. The main idea of this methodology is first to describe known profiles using a large number of control points, and then use NURBS curves to approximate this initial description. Because of the very large number of control points in the initial profile, shape modifications are awkward. Engineers thus use an optimization algorithm to simplify the NURBS used in the representation of a wing profile. They have tested this methodology with various target profiles (such as RAE2822, Boeing A8 and Bombardier-Canadair), and found that profiles can always be represented using a small number of control points, while maintaining the accuracy within to tolerances imposed by analysis and fabrication. Using the discrete approach, a profile may need to be represented by more than two hundred control points, however, if NURBS or B-Spline curves are used, the same profile can be descibed using only a

few tens of points (see the Figure 4.1), with a typical accuracy of $\epsilon_{max} \leq 8 \times 10^{-5}$ based on unitary chord length. This accuracy has been found sufficient to comply with manufacturing tolerance and flow solver sensitivity. Relying on fewer design parameters to represent the airfoils than before, NURBS representation reduces the number of the constraints involed in changing the wing shape, and makes the aerodynamic optimization process easier.

Airfoil shape optimization aims to minimize the number of control points of the NURBS which express a wing profile curve while maintaining sufficient precision. One important step is thus evaluating the error induced by the change of geometric representation going to NURBS. The optimization procedure is an iterative process, and at each iteration, the optimization process increases the number of control points and compute the approximation error between the current NURBS curve and the target. When the approximation error becomes small enough, the NURBS curve is the result that engineers will use in subsequent design phases.

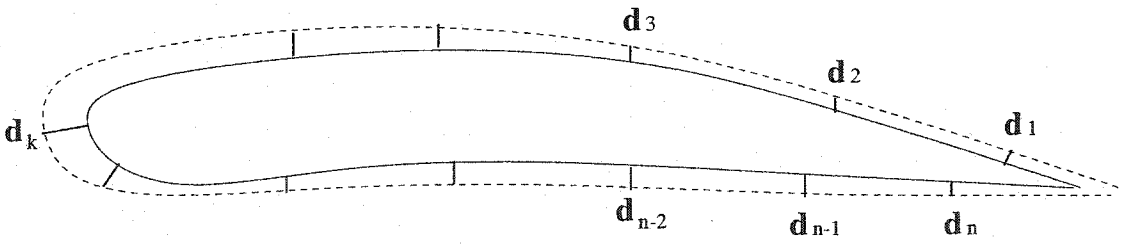


Figure 4.2 Check the approximation error of a NURBS

Sampling is used to calculate the approximation error. That is, along the new curve, many sample points are distributed according to the curvature of the profile (see the figure 4.2). The approximation error is calculated using a combination of the maximum distance and the average distance between the two curves. The following formulas are used:

$$\epsilon_{\text{avg}} = 1/n \sum_{k=1}^{n} d_k \tag{4.1}$$

$$\epsilon_{\text{max}} = \max_{1 \le k \le n} d_k \tag{4.2}$$

And then, the two distances are combined to evaluate the approximation error:

$$F(\mathbf{X}) = 2\ \epsilon_{\text{avg}} + \epsilon_{\text{max}} \tag{4.3}$$

In general, to accurately check any NURBS, a large number of sample points must be identified so that a stable evaluation of the maximum error ϵ_{max} and the average error ϵ_{avg} can be obtained. For example, the profile in Figure 4.2 requires at least five hundred sample points. Thus, the work load of calculation may become very heavy. To enhance calculation efficiency and simplify data management, an optimization procedure has been incorporated in the *VADOR* framework and will serve to evaluate the two distribution models.

4.1.2 Optimization process

The methodology for computing a geometric approximation with a NURBS proceeds as follows: the first step is interpolation, a B-spline is used to create a curve which has as many control points as the discrete profile. The second step is to approximate that curve, using a preselected number of control points, and this initial appoximation curves is provided as a starting point for the optimization process. The third step consists in using an optimization algorithm to refine the geometic representation of the objective airfoil profile by changing the number and position of the control points. This step aims to minimize the approximation error.

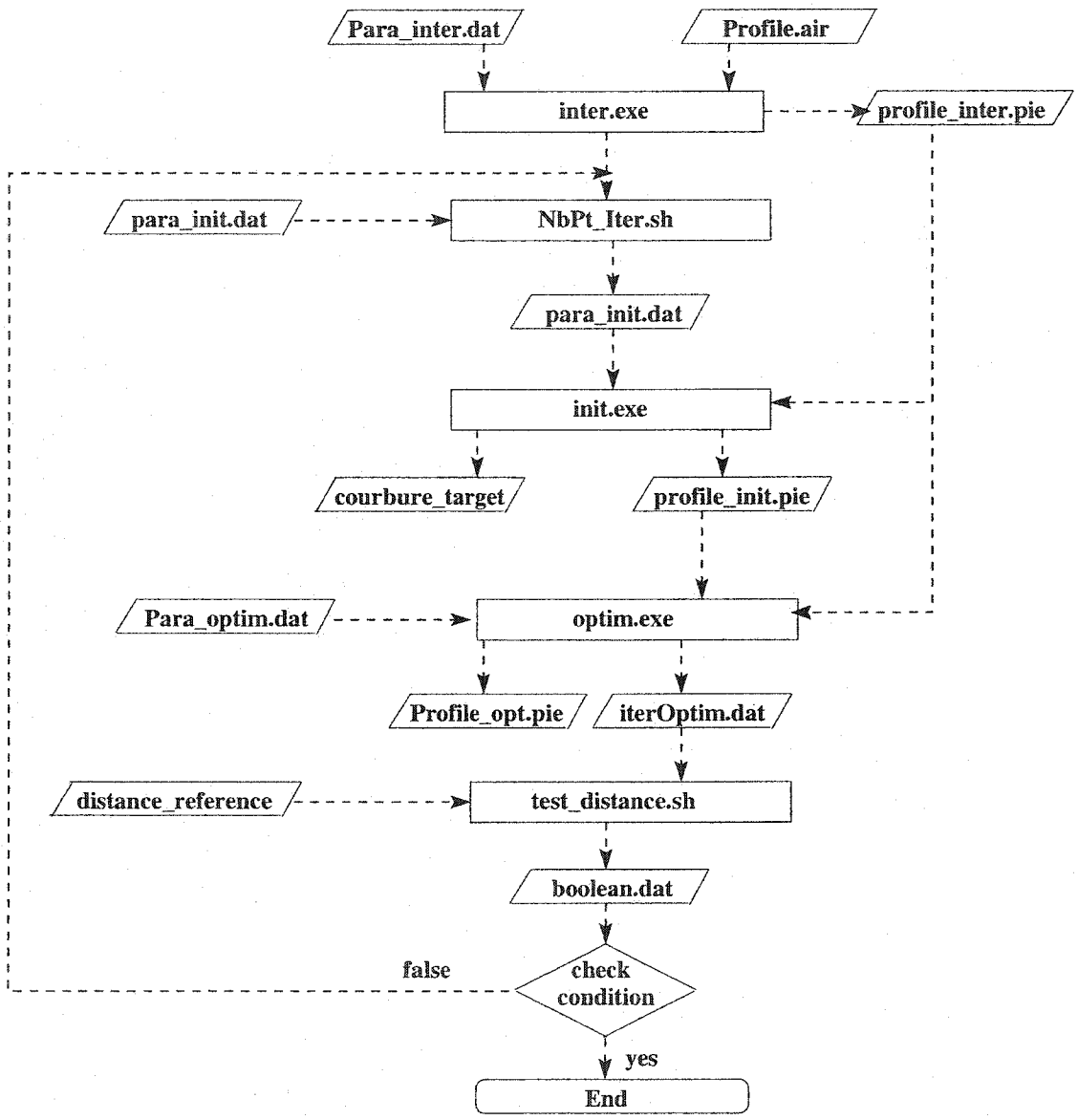


Figure 4.3 Profile approximation process

Figure 4.3 shows the execution flow chart of the approximation process. This task needs the cooperation of the CPU servers on one or two computers. All the computers are not in the same NFS domains.

The whole process begins with the invocation of the "inter.exe" application which builds the wing profile by interpolating a B-spline through the set of discrete input points. Its output file "profile_inter.pie" is the target B-spline curve which is represented by a large number of points. Next, the process enters a do-while loop, which is used to find a NURBS-based geometric approximation of the curve. For each iteration of the loop the number of the control points is changed so that the approximation profile moves closer to the target. This is done by the "NBPt_Iter.sh" script. Once the data of the control points is created, it is fed into the "init.exe" application, which calculates the geometric data of the new profile using NURBS approximation. The two profiles are then ready for optimization. A number of sample is selected to compute geometric points along the approximation curve and calculate the distance between the two profiles. This work is done by the "optim.exe" application. Finally, the data of the distance is sent to the "test_distance.sh" application to evaluate the error and determine convergence. The result of this application is a boolean value. If it is false, it means the error is still too large, and the process must start with another loop with a different number of control points and try again; on the other hand, if it is true, it means the approximation result is able to represent accurately the target profile and the overall process terminates.

In the VADOR project, the *Executive Server* and the *CPU* servers always complete a task by migrating the executable code (*MA* version) or static code (*C/S* version). This chapter will focus on how the *VADOR* project coordinate the Executive server and the *CPU* servers to execute a task, and compare the two approaches.

4.2 Description of test environment

To quantify the execution performance of each approach, two aspects have been considered. One is execution time. The execution time is computed as the elapsed time between the moment when the first command is sent from the *Executive server*, until the last "notifyFinish" signal is received from the *CPU server*. Execution time directly contributes to the perceived overall efficiency of the system, and constitutes one of the most important concerns of users. The second aspect considered is transaction volume, the amount of data that is transfered among the machines during the execution period. This factor is one measure that allows to determine the extent to which each approach relies upon large network bandwidth.

However, it is not easy to get pure testing results using a public network since there are many other applications running on the computers concurrently. Those applications consume resources of the machines and may exchange data through the network; the testing environment thus always changes, and execution time and transaction volume data can not be obtained accurately.

Therefore, to obtain precise test results, five personal computers have been connected together to form a small isolated local network on which all tests were run. Among these machines, one acted as a server and provided naming services and connection to the external network. It was identified as the "master". The other four machines were identical nodes of the network, named "node01" to "node04".

4.2.1 Hardware configuration

This test does not need any special configuration of computer hardware. The only requirement is that the machines be able to run Java programs and have access

to a network to exchange messages. Thus, except for the "master" which has an additional network card, all other machines had identical equipment. Here are the main characteristics of each machine:

CPU clock of main processor: 1.300GHz

Total memory: 256 MB

Capacity of hard disk: 40 GB

4.2.2 Software configuration

The *C/S* approach needs a Java runtime environment, and the *MA* approach needs both a Java runtime environment and a *MA* platform. Besides, a network monitor software is required to measure network usage of both approaches. The monitoring software should be able to count the number of data transactions and report results through a text file. Therefore, every machine is equipped with the following operating system and softwares:

Operating system: Redhat Linux 8.0

Java version: Java(TM) 2 Runtime Environment, Standard Edition 1.4.0_03

Mobile agent platform: Grasshopper v 2.2.4

Network monitor software: Argus v 2.0.5 (Argus network monitor, 2000)

Software for *C/S* approach: VADOR 1.2.2

Software of *MA* approach: VADOR Prototype for Mobile Agent

4.3 The test plan

As introduced before, each calculation of the airfoil shape optimization contains at least one loop, and the duration of the loop depends on the selection of the initial number of the control points. For the test profile chosen and selected level of accuracy, when the number of initial control points was set to 37 or more, only a single loop was needed. When the number was 36, two loops were required; when the number was 35, three loops were required, and so on. Inside each loop, there were five or six tasks that needed to be accomplished, and each task had its own set of input and output files. Often, the input files of a task were created by a previous task.

For every number of control points, four tests have been conducted. Two for the *C/S* approach, and two others for the *MA* approach. For each approach, one test was a simple test, which means that all data files (input and output) were stored on the same machine as the legacy applications. In this test, the network usage was relatively low since no data file needed to be transfered. The other test was a more complex test, which was set up so that every data file was located on a machine different from that of the legacy application. Hence, there needed to be a lot of downloading and uploading tasks to be performed by the *CPU server*.

Tests have been carried out as series comprising twenty initial numbers of control points have been tested. The *Executive server* is in charge of accumulating and reporting the execution time for each execution. When the first command of a test departed from the *Executive server*, the start time of the test was recorded. At the end of every loop, the *Executive server* checked the boolean result to determine if it needed to launch another loop. If not, it recorded the end time of the test and calculated the whole execution time. The same process happened for the network

monitoring software "Argus". When a test began, "Argus" was launched to capture any transaction on the network. After the test finished, "Argus" was terminated and output the records to a text file. An other Java program read the text file and gave out the final volume of transaction.

4.4 Execution without data file transfer

This test aimed to compare the performance of both approaches when only Java code was migrated among hosts. In this test, the machine named "node01" was selected to host the *Executive server*, the machine named "node03" was selected to host all the legacy applications, all the input and output files were also located on machine "node03", therefore, no data file transfer jobs affected execution time or transaction volume. The total execution time was only the duration needed by the programs to process all the commands and execute the legacy applications. The total network transaction volume is the static configuration data (in *C/S* approach) or the executable code (in *MA* approach) that was migrated between two machines.

4.4.1 Description of the test using the Client/Server model

This test was conducted between the *Executive server* which was deployed on the machine "node01" and the *CPU server* which was deployed on the machine "node03". The *Executive server* sequentially sent commands to the *CPU server* and checked the boolean result after each loop.

4.4.2 Description of the test using the Mobile Agent model

This test was conducted between the machine "node03" and the machine "node01" which was equipped with the *Executive server* and the code of the *CPU server*. Besides, both of the machines had a mobile agent platform, Grasshopper, running. For each task, since no data file was transfered, the *Executive server* only sent two agents, one commander agent and one soldier agent which was in charge of executing a legacy code, to the machine "node03". Table I.2 shows the execution time of every loop.

4.4.3 Result of the test

Figure 4.4 compares the execution time results obtained using both approaches. Numerical values are presented in Table I.1 and Table I.2 in the appendix.

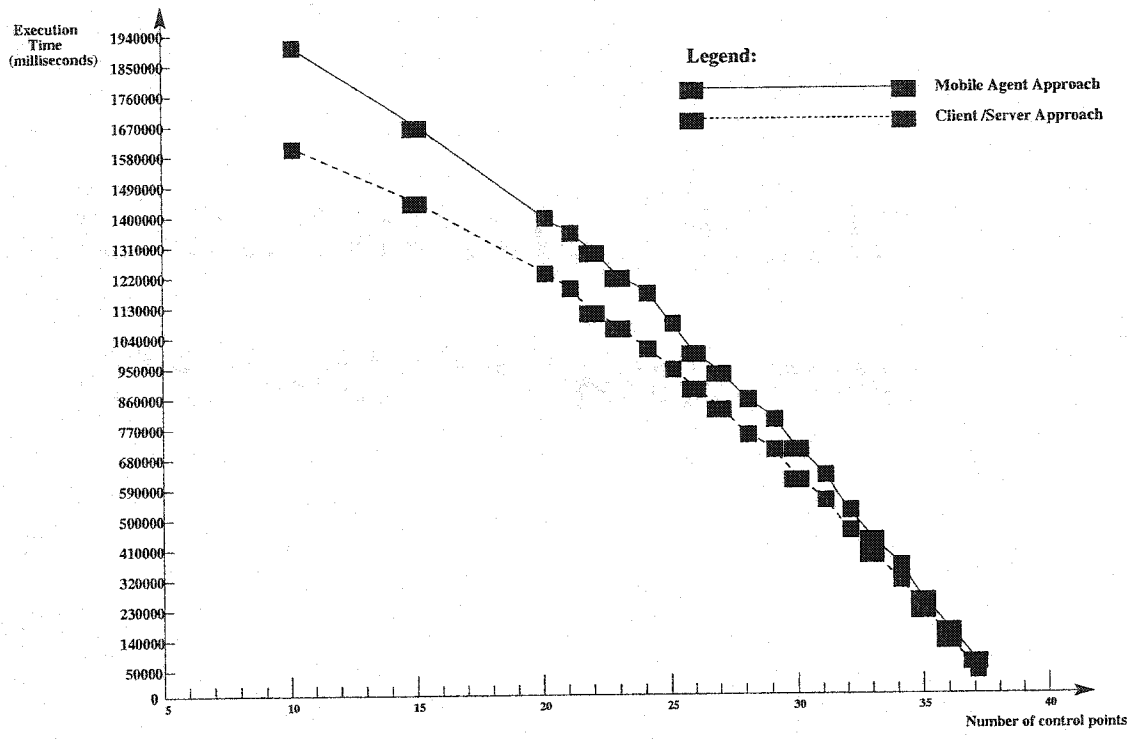


Figure 4.4 Comparison of Execution Time (no data file transfer)

Figure 4.5 compares the transaction volume results obtained using both approaches. Numerical values are presented in Table I.5 and Table I.6 in the appendix.

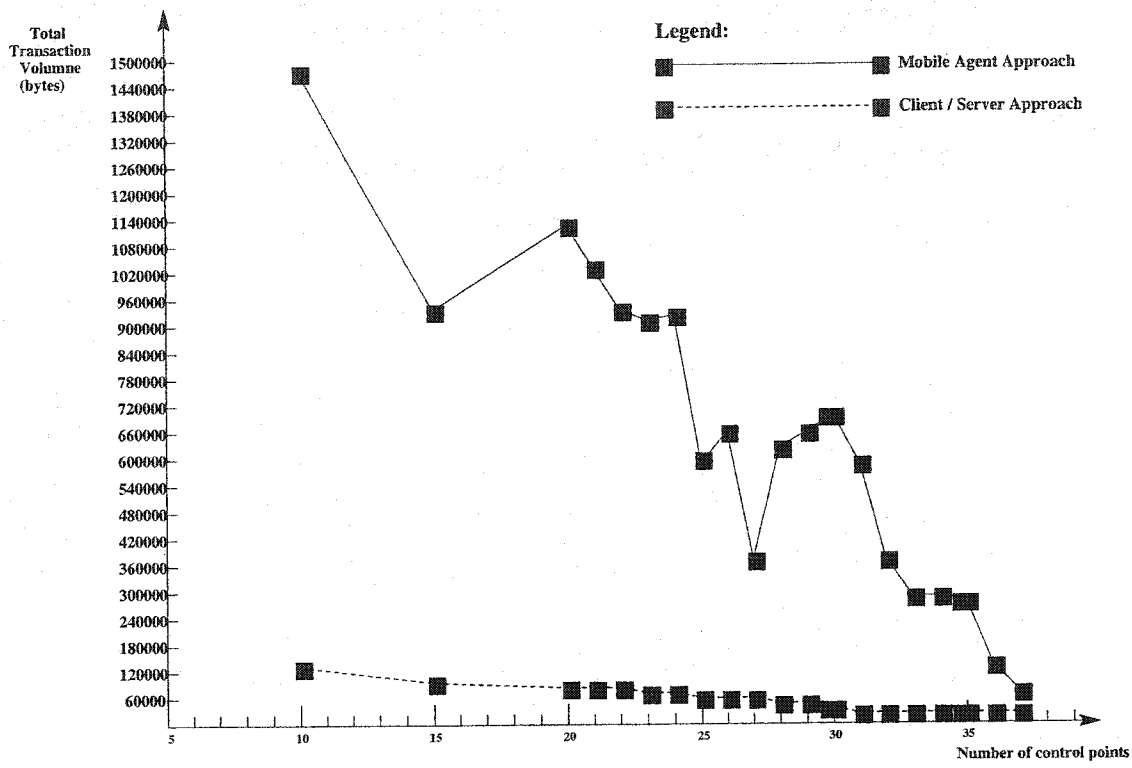


Figure 4.5 Comparison of Transaction Volume (no data file transfer)

4.5 Execution with data file transfer

Different from the section 4.4, the test in this section put all the input and output data files on the machine "node01" rather than "node03". Therefore, every task had to download all the necessary files from the remote computer before it began the execution, and then had to send all the result data files back to the remote machine at the end. Obviously, the workload for this test was heavier than for the test in the section 4.4. We can also compare the performance of both approaches from an other respect – complexity of processing.

4.5.1 Description of the test using the Client/Server model

This test was conducted between the *Executive server* which was deployed on the machine "node01" and the *CPU server* which was deployed on the machine "node03". The *Executive server* sequentially sent commands to the *CPU server* and checked the boolean result after each loop.

4.5.2 Description of the test using the Mobile Agent model

This test was conducted between the machine "node03" and the machine "node01" which was equipped with the *Executive server* and the code of the *CPU server.* Besides, both of the machines had a mobile agent platform, Grasshopper, running. To accomplish each task, the *Executive server* deployed several agents to the machine "node03": one commander agent to coordinate the processing, one soldier agent to execute the legacy code, one soldier agent which was in charge of performing file management, and one soldier agent to download the necessary input files. Besides, an other soldier agent was delpoyed on the machine "node03" to

complete the uploading job. For each input and output data file, the soldier agent for uploading and downloading tasks cloned itself so that all possible transactions were handled in parallel.

4.5.3 Result of the test

Figure 4.6 compares the execution time results obtained using both approaches. Numerical values are presented in Table I.3 and Table I.4 in the appendix.

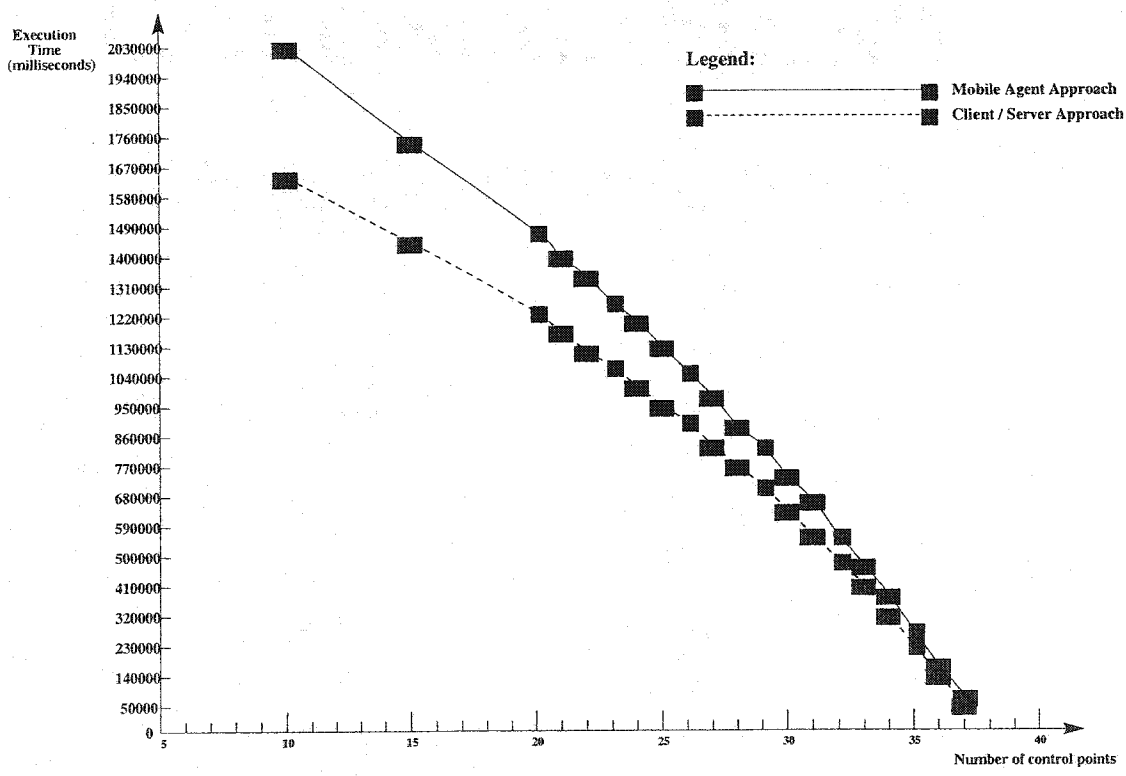


Figure 4.6 Comparasion of Execution Time (with data file transfer)

Figure 4.7 compares the transaction volume results obtained using both approaches. Numerical values are presented in Table I.7 and Table I.8 in the appendix.

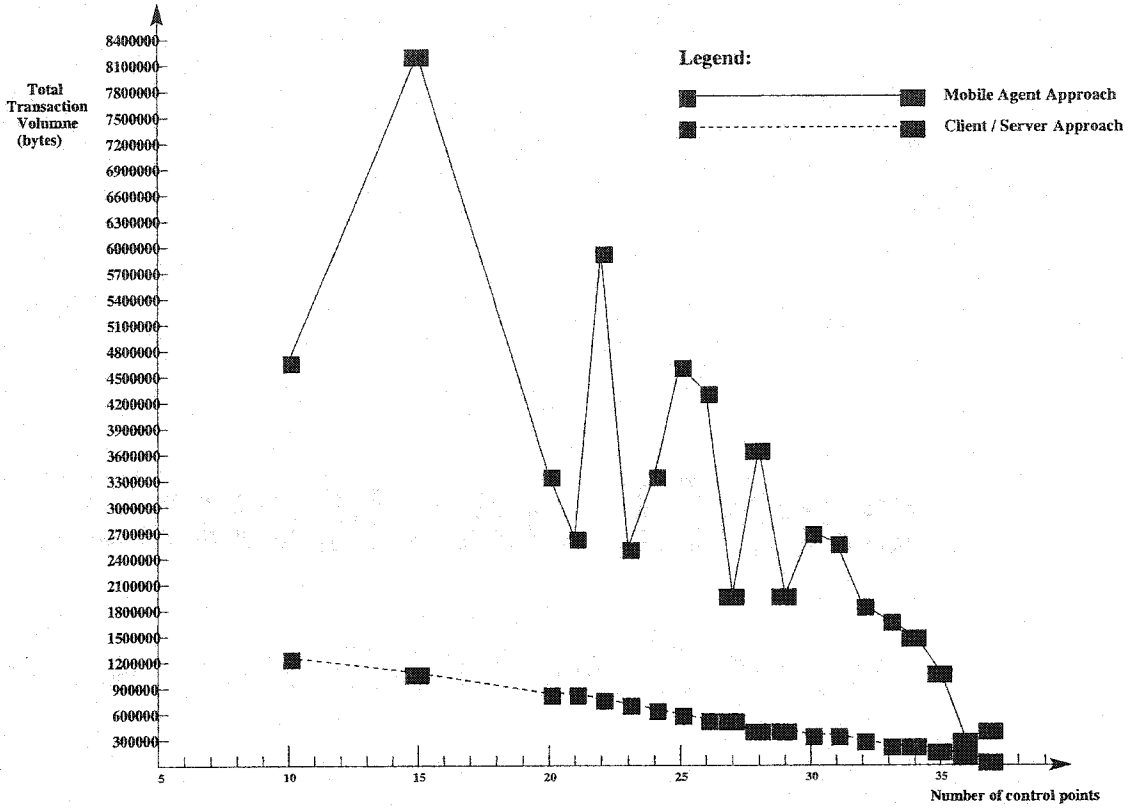


Figure 4.7 Comparasion of Transaction Volume (with data file transfer)

4.6 Comparison of the two approaches

In the previous two sections, both of the two approaches, *C/S* and *MA*, processed the same tasks under the same environment. The test results show that their performances are very different. In this section, we will summarize the differences of the two approaches from several respects so that we can get a clear sense of their

advantages and disadvantages.

4.6.1 Transfering task information

When the Executive server wants to utilize the resources of a machine to execute a legacy application, it must migrate a piece of code which carries the information of the task to that computer. If the CPU server is in *C/S* model, the first thing is setting up a big vector which contains the information of all the sub-tasks. When this vector is sent to the target machine, it is read and translated into a series of jobs. The main purpose of this encapsulation/interpretation process is to make sure that all the information can be transfered in one transaction. However, if the CPU server is in *MA* model, this process is omitted. Since the code of the CPU server is deployed at the Executive server side, the request from the Executive server is directly translated into a job package and assigned to the commander agent. This method reduces the workload of programming, and makes the transfer process more efficient.

4.6.2 Controling task execution process

As discussed before, the CPU server uses a "callback" principle to manage the whole task execution process. In the *C/S* model, the CPU server prepares a big agency to treate each step of the execution. The agency includes three parts: "TaskControler","ResultControler" and "ThreadControler". Every job must reserve a place in the "ResultControler" when it starts to work, and register its result when it finishes. If this job launches a local or remote thread, it also must log on to or out of the thread to the "ThreadControler". For the *MA* CPU server, this process becomes a little bit simpler. Every job is taken by a soldier agent. The developers

do not need to take care of the management of the threads, or the style of the task (long-time or short-time). The commander agent only coordinates its soldiers, keeps waiting for the results, and decides on the next proper behavior. Thus, to manage a task, the code of the *MA* model is more concise than that of the *C/S* model.

4.6.3 Deployment and maintainance of the Vador code

Currently, the VADOR project uses the *C/S* model. When it is installed on a system, every machine should get a copy of the CPU server and start it running. This model can work robustly. However, if maintainers want to update the *CPU server* code to a new version, they will find the *C/S* deployment approach less convenient because they will need to change the copies of the code on each machine and restart the servers. The *MA* model does not have this kind of worry because there is only one copy of the *CPU server* code installed, the updating process thus becomes much simpler. Moreover, since the *MA* model simplifies the code of task management, the development of the programs is simplified. For example, if a new type of request needs to be added to the Vador project, in the *C/S* model, the developer must modify the "CommandBuilder" to embed the request into the command vector, and modify the "CommandInterpretor" to create the related task. In the *MA* model, since the jobs are deployed only at the *Executive server* side, the developer only modifies the definition of one agent to add the new kind of task.

On the other hand, the characteristics of the *MA* model limits its functionality. Since the agent always migrates among the computers, its size should be considered carefully. Therefore, an agent can not contain too many functions. To solve this problem, in the *MA* CPU server, all the functions are distributed into several agents. For each task, the CPU server only selects the necessary agents needed for

the work. However, the *C/S* version of the CPU server does not care about its size because all the programs are located at the local host and will not move. It is therefore easier for the developers to expand the functions of the CPU server.

4.6.4 Network burden

Both of the two approaches need migrating some form of code to remote machines. The *C/S* version of *CPU server* only migrates some configuration data contained in vectors which get interpreted by the *CPU server*. The *MA* version of CPU server migrates not only the variables, but also the executable code. This characteristic causes more burden on the network. Let us examine the figures of the data transaction volume of the two approaches to compare their respective network burdens:

In the absence of data file transfer (Figure 4.5), the network burden of the *MA* approach is more than 10 times larger than the *C/S* approach. If all the data files need to be transfered (Figure 4.7), the network burden of the *MA* approach is still about 5 times larger than the *C/S* approach. Typical use of the VADOR framework for actual engineering tasks can be estimated to lie roughly in the middle of those two extremes, and the burden of the *MA* approach could thus be estimated to represented between 5 and 10 times the burden of the *C/S* approach.

Besides, some useless functions in an agent also cause additional network burden. For example, in the agent "SoldierFileMA", there are four functions: building a temporary directory, copying a file, changing the group name of a file and removing a file or a directory. Sometimes, the function for copying a file is not used on the target platform, but the code of this function still accompanies the agent and occupies a certain bandwidth on the network.

4.6.5 Efficiency of processing

Currently, the *C/S* approach is a little bit more efficient than the *MA* approach. This difference may be related to the fact that the *C/S* approach does not need to process the migration of agents. Normally, the execution time is an important criterion in the of evaluation of the efficiency of an approach. Figure 4.4 and 4.6 show the difference of execution time of both approaches. We can see that total execution time of the two approaches are similar, with a small consistent advantage for the *C/S* approach.

4.6.6 Platform support

In this respect, the *C/S* model has strong advantages over the *MA* model. From the description of the test environment in section 4.2, we can see that the *C/S* approach only needs a Java runtime environment which has already become a standard configuration on most operating systems. The *MA*, however, needs a mobile agent platform (for example, Grasshopper, Concordia, Voyager, etc) running on every machine. Currently, the communication between different platforms is not uniformized, the agents on different platforms can not exchange messages or even migrate at will.

Besides, the stability and maturity of the *MA* platforms is still very weak. In the tests presented in this chapter, if the number of the control points is less than 10, the Grasshoppper can not work normally because there are too many agents that need to visit the same *MA* platform and this makes the platform crash. However, under the same conditions, the *C/S* approach can work very well. Moreover, please compare the Figures of the test results: in the Figures 4.5 and 4.7, the curves which descibe the data transaction volume of the *MA* approach are very tortuous,

whereas the curves that descibe the data transaction volume of the *C/S* approach are smooth and regularly going down. And, as mentioned in the section 4.6.4 and 4.6.5, the *MA* approach consumes much more resources than the *C/S* approach. The reason of this difference is that the technology of the *MA* platform is not stable and mature yet. This is also the reason that no *MA* platform is selected to be a common equipment of any operating system. There still seems to remain lots of work to be accomplished in order to fully take advantage of the possible performance benefits promised by *MA* technology.

CHAPTER 5

CONCLUSION

The *VADOR* project provides a uniform and structured analysis environment for the aeronautic engineers, so that they can define and run analysis procedures and share the result data files with their partners within or across departments. *VADOR* has two purposes. The first one is to integrate the management of all the legacy analysis applications, help them communicate with each other and share information. The other one is to create an efficient environment which allows to implement Muti-Disciplinary Optimization (MDO) practices. In order to utilize distributed resources to accomplish engineering computations, *VADOR* migrates some pieces of executable code to the target computers which can provide the necessary services. Currently, there are two models of code migration: the *C/S* model and the *MA* model. Based on the analysis of the two strategies, we have been able to get a clear sense of the advantages and disadvantages of the two models, and be able to make a recommandation as to which distribution approach provides the best expectancy of success for a production implementation of the *VADOR* framework.

In the *C/S* model, two programs (Client and Server) communicate with each other to exchange the necessary informations. Usually, the two parts are on different computers, but their functionalities are not fundamentally different. If necessary, they can switch their roles in the model. Every machine can be both a Client and a Server. This model provides a good chance that the distributed computers can efficiently utilize the resources of the global system. From the aspect of architecture, this model is classified through the concept of tier, the most popular architectures

being two tiers and three tiers.

The Client in the two tiers architecture only takes care of the interaction with the users, the Server side carries out all other issues, such as keeping the data, processing the requests, etc. This architecture has very clear duties for each part, thus it is easy for the developers to set up a modular design. This design is convenient for implementing uniform business rules for homogeneous applications on the Client side. However, since the Server must handle both the work of business logic and the interaction with the DBMS, the interoperability and scalability of this architecture is limited. The three tiers architecture improves the structure at the Server side. It sets up a middle tier which manages the business logic so that the Server can be liberated from heavy computation works and can focus on the database access or resource utiliztion. Since the middle tier hides many complexities of underlying services, the flexibility, scalability and performance of the total architecture is increased. Its main disadvange is that the process of developing this architecture is more complicated than the two tiers one.

So far, the *C/S* model has been widely used in the world. Many technologies are based on it. If a software engineer wants to develop a project which is based on this model, the first thing is to compare the advantages and disadvantages of the various architectures. Generally, if the structure of the software is not very complex, or the data transaction is not too heavy, two tiers architecture is a good choice; if the Server side wants to manipulate heterogeneous databases and processing rules,or the system needs to carry out many requests from the users simultaneously, three tiers architecture should be selected. For example, the *VADOR* project can be divided into three tiers. The presentation tier (Client tier) is taken by the *VADOR GUI*, the *VADOR* users express their ideas by clicking mouse buttons on the screen. Then the requests are sent to the middle tier – the *Executive server*.

The *Executive server* identifies the purposes of the users and balances the workload on every machine, and then sends requests to the third tier: the Librarian server which takes care of the database access, and the *CPU server* which provides the calculation services. However, if we only focus on the relationship between the *Executive server* and the *CPU server*, it is a two tiers architecture because the *CPU server* concentrates only on one type of work: using the resources of the local machine to provide calculation services.

The *MA* model is a relative new technology for distributed systems. When a Client wants to utilize the resources of other computers, it sends out a piece of executable code (agent) to travel through the system. An agent has two main characteristics, one is autonomy, which means it can plan and control its own behavior in order to acheive the final goal; the other is mobility, which enable the agent to move from one machine to another. Numerous ongoing projects are being developed to build platforms which support the execution of mobile agents. From the aspect of mobility, the agents can be sorted into three groups : the first group has constrained mobility. In this case, the unique destination of the agents are specified by the Client side program. After the agents finish their jobs on that machine, they will not plan to migrate to other places. The advantage of this kind of agent is its conciseness and simplicity, but it lacks in autonomy. The second group has weak mobility. The agents only carry some initialization data with their next trip, and do not care about their current executing stage. This kind of agent is autonomous enough to accomplish its mission while it is migrating among a set of computers. Its only disadvantage is that when it gets to a new machine, it must start its work from the beginning rather than from the executing point it had reached on the previous host. The last group has strong mobility. The migration of this kind of agents is really transparent. The agents can temporarily suspend their execution process and move to another machine to resume their mission. However, since they

always carry the information of the executing point, their size is bigger than the previous two kinds, and their architecture is more difficult to implement.

In order to fully take advantage of the possibilities offered by *MA* technology, communication among agents should be standardized. To allow communication among agents and with their environments – ACL. Currently, there are lots of *MA* systems in existence, they all have their own ACLs. Therefore, an urgent issue must be considered: how to uniformize those ACLs so that the heterogenous agents and platforms can interact with each other without any barrier. After several years of effort, people now mainly focus on three ACLs: KQML, FIPA and MASIF. The FIPA ACL seems more promising than the other two, because it has enough interoperablity and scalability. Besides, FIPA ACL is produced by a well-organized standard body, OMG. To maintain FIPA ACL, this organization has developed sufficient formal documentation. This ACL is being accepted by more and more projects.

In the course of this work, we have studied and compared the two most common architectures for code distribution, namely the three-tier *C/S* architecture and the weakly mobile agents of the Grasshopper platform. The analysis of each approach has been based on a careful evaluation of distribution issues pertinent to each, which has lead to a working implementation of the *CPU server* for both distribution models and their integration in the *VADOR* framework. While the *MA* version of the *CPU server* remains in a prototype state, the *C/S* version has been sufficiently developped to enable deployment of the framework at Bombardier Aerospace.

Both version of the *CPU server* face the same challenges: how to accurately transfer the requests of the users to the target machines, how to arrange the time schedule of all the threads, how to decrease the workload on the network, and maximize program efficiency. In the *C/S* model, all task informations are encapsulated into

a command package, and are delivered to the target machine together. At the *CPU server* side, the command package is translated into several jobs which are finally assigned to some threads to be run. To coordinate the threads, the *C/S* version of the *CPU server* maintains a "call-back" mechanism which can manage the results of all the jobs, and decide on the next step of work according to the registration of each result. However, the *MA* version of the *CPU server* uses a different method to accomplish the same task. It interprets the command and prepares the jobs at the *Executive server* side, launches several agents, and sends them to the target machine after individually assigning jobs to them. On the remote machine, the commander distributes the proper jobs to its soldiers and monitors their working process until the jobs are successfully finished.

From some aspects, the *MA* version has advantages over the *C/S* version. At first, it avoids the "bottle-neck" effect, since the code of this approach is deployed at the same host as that of the *Executive Server*, the process of encoding the requests into vectors and interpreting the vector into requests is omitted. If a new type of request needs to be transfered, the modification of this approach is simplier than the *C/S* version. Second, the agents which are sent to the target machine are the representatives of the *Executive server*, they can autonomously supervise the working process of the task, do not need to set up a bulk of data structures to contain the temporary results. Besides, the *MA* version only keeps one copy of the code in a system, so it is easier to update. Although the *MA* version has those merits, it doesn't mean that the *C/S* version is not useful any more. On the contrary, its speciality still attracts people's attention. Since it has been experimented with for many years, its architecture and theory is mature. It can effectively reduce the traffic on the network, and because it does not need to consider migrating its programs, the volume of code on both sides can be large enough to carry very complicated missions. Up to now, the utilization of the *MA* technology is limited

by the bandwidth of the current networks and the standardization issues of the platforms. In the future, when the *MA* systems become sufficiently mature and are embeded into every operating system as a common equipment, people will be able to fully enjoy the convience that this model brings.

REFERENCES

ARGUS NETWORK MONITOR (2000). http://qosient.com/argus.

BAUMANN, J., HOHL, F., STRABER, M. et ROTHERMEL, K. (1997). Mole concepts of a mobile agent system. http://ncstrl.informatik.uni-stuttgart.de/Dienst/UI/2.0/Describe/ncstrl.ustuttgart_fi/TR-1997-15.

BETTINI, L. (2001). Translating strong mobility into weak mobility. *Proc. of 5th IEEE Int. Conf. on Mobile Agents (MA).*

BREWINGTON, B. (1999). Mobile agents in distributed information retrieval. *Intelligent Information Agents.*

FUGGETTA, A. (1998). Understanding code mobility. *IEEE Transactions on software engineering.*

GRAY, R. S. (1997). Agent tcl: A flexible and secure mobile-agent system. *Fourth Annual Tcl/Tk Workshop (TCL 96).*

HOLGER PEINE, T. S. (1997). The architecture of the ara platform for mobile agents. *First International Workshop on Mobile Agents MA'97.*

IKV++ TECHNOLOGIES AG (2002). Grasshopper. http://www.grasshopper.de/.

JAVA WORLD (2000). One, two, three, or n tiers? http://www.javaworld.com/javaworld/jw-01-2000/jw-01-ssj-tiers-p4.html.

LANGE, D. B. (1998). *Programming and Deploying Java Mobile Agents With Aglets.* Addison-Wesley Pub Co.

MICROSOFT COORPERATION (2000). Com. http://www.microsoft.com/com/tech/com.asp.

NDIAYE, A., TRÉPANIER, J.-Y., GUIBAULT, F., OZELL, B. et MAHDAVI, B. (2000). Database requirements for an mdo software framework. *CFD2K, 8e conférence annuelle de la Société canadienne de CFD, Montréal, 11 au 13 juin.*

OMG GROUP (1997). Corba. http://www.corba.org/.

OMG GROUP (1998). MASIF-RTF results. http://www.fokus.gmd.de/research/cc/ecco/masif/body_documents.html.

OMG GROUP (2001). Fipa abstract architecture specification. *Foundation For Intelligent Physical Agents.*

RECURSION SOFTWARE INC. (2001). Voyager mobile agent system. http://www.recursionsw.com/products/voyager/voyager.asp.

RSA SECURITY (2002). Two-factor authentication for a mobile world. http://www.rsasecurity.com/products/mobile/.

SOFTWARE ENGINEERING INSTITUTE (SEI) (1997a). Client/server software architectures–an overview. http://www.sei.cmu.edu/str/descriptions/clientserver_body.html.

SOFTWARE ENGINEERING INSTITUTE (SEI) (1997b). Three tier software architectures. http://www.sei.cmu.edu/str/descriptions/threetier_body.html.

SOFTWARE ENGINEERING INSTITUTE (SEI) (1997c). Two tier software architectures. http://www.sei.cmu.edu/str/descriptions/twotier_body.html.

STANFORD (1997). http://piano.stanford.edu/concur/language.

SUN COORPERATION (2002). Enterprise java bean. http://java.sun.com/products/ejb/.

THE APACHE SOFTWARE FOUNDATION (1996). http://www.apache.org/.

THE FUNDATION OF INTELLIGENT PHYSICAL AGENT (2000). Fipa specification. http://www.fipa.org/specifications/index.html.

THE OPEN GROUP (2000). The OSF distributed computing environment. http://www.osf.org/dce/.

TIM FININ, R. F. (1994). Kqml - a language and protocol for knowledge and information exchange. http://www.cs.umbc.edu/kqml/papers/kbkshtml/kbks.html.

TRÉPANIER, J., GUIBAULT, F., OZELL, B. et PELLETIER, D. (2000). Vador. http://www.cerca.umontreal.ca/vador.

TRIGON BLUE INC. (2000). Desktop and file sharing architecture. http://trigonblue.com/file_sharing.htm.

VITEK, J. (1997). Sumatra: A language for resource-aware mobile programs. *Mobile Object Systems: Towards the Programmable Internet.*

WAYNE JANSEN, T. K. (1999). NIST special publication 800-19: Mobile agent security. *NIST Special Publication 800-19: Mobile Agent Security.*

YOUNG, A. (1997). Sliding encryption: A cryptographic tool for mobile agents. *Sliding Encryption: A Cryptographic Tool for Mobile Agents.*

APPENDIX I

Execution data results

Table I.1 Execution time of the Client/Server model (no data file transfer)

Number of control points	Total execution time
10 points	1619855 milliseconds
15 points	1446026 milliseconds
20 points	1228831 milliseconds
21 points	1178139 milliseconds
22 points	1122246 milliseconds
23 points	1069052 milliseconds
24 points	1014876 milliseconds
25 points	951229 milliseconds
26 points	894109 milliseconds
27 points	828647 milliseconds
28 points	766083 milliseconds
29 points	700711 milliseconds
30 points	632949 milliseconds
31 points	562907 milliseconds
32 points	482076 milliseconds
33 points	405378 milliseconds
34 points	315408 milliseconds
35 points	232423 milliseconds
36 points	144098 milliseconds
37 points	57980 milliseconds

Table I.2 Execution time of the Mobile Agent model (no data file transfer)

Number of control points	Total execution time
10 points	1903900 milliseconds
15 points	1678118 milliseconds
20 points	1404068 milliseconds
21 points	1357625 milliseconds
22 points	1282766 milliseconds
23 points	1222746 milliseconds
24 points	1158058 milliseconds
25 points	1082083 milliseconds
26 points	1013533 milliseconds
27 points	936493 milliseconds
28 points	867386 milliseconds
29 points	790320 milliseconds
30 points	711611 milliseconds
31 points	636973 milliseconds
32 points	540664 milliseconds
33 points	456740 milliseconds
34 points	359570 milliseconds
35 points	264311 milliseconds
36 points	167586 milliseconds
37 points	70157 milliseconds

Table I.3 Execution time of the Client/Server model (with data file transfer)

Number of control points	Total execution time
10 points	1635889 milliseconds
15 points	1448144 milliseconds
20 points	1227764 milliseconds
21 points	1179630 milliseconds
22 points	1121897 milliseconds
23 points	1069919 milliseconds
24 points	1019611 milliseconds
25 points	952376 milliseconds
26 points	894807 milliseconds
27 points	830798 milliseconds
28 points	765310 milliseconds
29 points	701767 milliseconds
30 points	633805 milliseconds
31 points	564889 milliseconds
32 points	482249 milliseconds
33 points	406803 milliseconds
34 points	317362 milliseconds
35 points	232820 milliseconds
36 points	146246 milliseconds
37 points	58562 milliseconds

Table I.4 Execution time of the Mobile Agent model (with data file transfer)

Number of control points	Total execution time
10 points	2021193 milliseconds
15 points	1756443 milliseconds
20 points	1480133 milliseconds
21 points	1399807 milliseconds
22 points	1344229 milliseconds
23 points	1273585 milliseconds
24 points	1204796 milliseconds
25 points	1136849 milliseconds
26 points	1056135 milliseconds
27 points	980575 milliseconds
28 points	904061 milliseconds
29 points	829078 milliseconds
30 points	740752 milliseconds
31 points	658231 milliseconds
32 points	562396 milliseconds
33 points	474816 milliseconds
34 points	375256 milliseconds
35 points	281726 milliseconds
36 points	176987 milliseconds
37 points	73844 milliseconds

Table I.5 Transaction volume of the Client/Server model (no data file transfer)

Number of control points	node03 -> node01	node01-> node03	Total transaction
10 points	36016 bytes	94247 bytes	130263 bytes
15 points	29642 bytes	77621 bytes	107263 bytes
20 points	23070 bytes	60905 bytes	83975 bytes
21 points	21782 bytes	57569 bytes	79351 bytes
22 points	20560 bytes	54233 bytes	74793 bytes
23 points	19338 bytes	50909 bytes	70247 bytes
24 points	18116 bytes	47585 bytes	65701 bytes
25 points	16828 bytes	44279 bytes	61107 bytes
26 points	15672 bytes	40913 bytes	56585 bytes
27 points	14186 bytes	37643 bytes	51829 bytes
28 points	12964 bytes	34229 bytes	47193 bytes
29 points	11812 bytes	30893 bytes	42705 bytes
30 points	10396 bytes	27557 bytes	37953 bytes
31 points	9178 bytes	24167 bytes	33345 bytes
32 points	8026 bytes	20885 bytes	28911 bytes
33 points	6676 bytes	17603 bytes	24279 bytes
34 points	5458 bytes	14225 bytes	19683 bytes
35 points	4174 bytes	10877 bytes	15051 bytes
36 points	2758 bytes	7529 bytes	10287 bytes
37 points	1606 bytes	4259 bytes	5865 bytes

Table I.6 Transaction volume of the Mobile Agent model (no data file transfer)

Number of control points	node03 -> node01	node01-> node03	Total transaction
10 points	374698 bytes	1105768 bytes	1480466 bytes
15 points	25880 bytes	910832 bytes	936712 bytes
20 points	412941 bytes	714413 bytes	1127354 bytes
21 points	352464 bytes	674993 bytes	1027457 bytes
22 points	309077 bytes	635878 bytes	944955 bytes
23 points	316347 bytes	596178 bytes	912525 bytes
24 points	377523 bytes	557661 bytes	935184 bytes
25 points	79652 bytes	518753 bytes	598405 bytes
26 points	181925 bytes	479973 bytes	661898 bytes
27 points	82359 bytes	294175 bytes	376534 bytes
28 points	224506 bytes	400660 bytes	625166 bytes
29 points	408411 bytes	254816 bytes	663227 bytes
30 points	375994 bytes	321923 bytes	697917 bytes
31 points	301823 bytes	282576 bytes	584399 bytes
32 points	162860 bytes	205531 bytes	368391 bytes
33 points	75061 bytes	205540 bytes	280601 bytes
34 points	120432 bytes	165654 bytes	286086 bytes
35 points	144283 bytes	127049 bytes	271332 bytes
36 points	71539 bytes	48880 bytes	120419 bytes
37 points	1474 bytes	5070 bytes	6544 bytes

Table I.7 Transaction volume of the Client/Server model (with data file transfer)

Number of control points	node03 -> node01	node01-> node03	Total transaction
10 points	547345 bytes	740165 bytes	1287510 bytes
15 points	452188 bytes	614257 bytes	1066445 bytes
20 points	357427 bytes	486831 bytes	844258 bytes
21 points	337695 bytes	460597 bytes	798292 bytes
22 points	318482 bytes	434897 bytes	753379 bytes
23 points	299411 bytes	408517 bytes	707928 bytes
24 points	279607 bytes	382948 bytes	662555 bytes
25 points	259985 bytes	356242 bytes	616227 bytes
26 points	240020 bytes	329914 bytes	569934 bytes
27 points	220822 bytes	304102 bytes	524924 bytes
28 points	201469 bytes	277926 bytes	479395 bytes
29 points	181503 bytes	251295 bytes	432798 bytes
30 points	161944 bytes	224978 bytes	386922 bytes
31 points	142416 bytes	197773 bytes	340189 bytes
32 points	122117 bytes	171135 bytes	293252 bytes
33 points	102751 bytes	144279 bytes	247030 bytes
34 points	83028 bytes	117366 bytes	200394 bytes
35 points	63246 bytes	90287 bytes	153533 bytes
36 points	43233 bytes	63053 bytes	106286 bytes
37 points	23272 bytes	35950 bytes	59222 bytes

Table I.8 Transaction volume of the Mobile Agent model (with data file transfer)

Number of control points	node03 -> node01	node01-> node03	Total transaction
10 points	2133540 bytes	2624542 bytes	4758082 bytes
15 points	6097215 bytes	2160392 bytes	8257607 bytes
20 points	1654072 bytes	1701150 bytes	3355222 bytes
21 points	1600744 bytes	1050006 bytes	2650750 bytes
22 points	4295699 bytes	1516239 bytes	5811938 bytes
23 points	1104892 bytes	1421240 bytes	2526132 bytes
24 points	1995780 bytes	1330693 bytes	3326473 bytes
25 points	3453169 bytes	1238132 bytes	4691301 bytes
26 points	3219357 bytes	1145544 bytes	4364901 bytes
27 points	919614 bytes	1054178 bytes	1973792 bytes
28 points	2679425 bytes	960613 bytes	3640038 bytes
29 points	1105832 bytes	866193 bytes	1972025 bytes
30 points	1946960 bytes	772145 bytes	2719105 bytes
31 points	1910026 bytes	681262 bytes	2591288 bytes
32 points	1252187 bytes	587972 bytes	1840159 bytes
33 points	1218378 bytes	492884 bytes	1711262 bytes
34 points	1110895 bytes	401100 bytes	1511995 bytes
35 points	667198 bytes	306845 bytes	974043 bytes
36 points	98095 bytes	214589 bytes	312684 bytes
37 points	326949 bytes	120160 bytes	447109 bytes